MASTER EN SCIENCES INFORMATIQUES

Evaluation de performance et amélioration d'une implémentation du protocole BGP-4

Tandel, Sébastien

Award date:
2002

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2001 - 2002

Evaluation de performance et amélioration d'une implémentation du protocole BGP-4

Sébastien Tandel

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique

Résumé

Ce mémoire entreprend de réaliser une évaluation des performances d'une implémentation du protocole de routage inter-domaine BGP-4 (Border Gateway Protocol) pour ensuite effectuer une amélioration. En général, améliorer les performances d'une implémentation est toujours intéressante. Cependant, l'amélioration d'une implémentation de ce protocole est d'autant plus intéressante qu'il subsiste encore à l'heure actuelle des discussions sur les problèmes engendrés par BGP parfois appelé malicieusement Border Glue Protocol.

Le protocole BGP-4 est le protocole qui permet de construire les tables de routage nécessaires au forwarding dans l'Internet global. Ce protocole s'est imposé comme le standard *de facto*. Pourtant, il nécessite encore des améliorations pour pouvoir subsister, du moins à moyen terme, comme protocole de routage inter-domaine.

Ce mémoire est donc principalement axé sur le protocole BGP-4 et une implémentation de celui-ci, *Zebra*. Ce mémoire se propose d'examiner les performances de cette implémentation. Ces évaluations permettront de souligner un point faible de cette implémentation que ce mémoire se propose d'améliorer.

Deux versions de l'amélioration seront envisagées et implémentées. Celles-ci seront testées de la même manière que la version originale de l'implémentation. Finalement, des tests plus poussés seront effectués afin de pouvoir distinguer les caractéristiques de chaque version de l'implémentation qui amèneront à des conclusions plus correctes sur les réelles performances des implémentations.

mots-clés : performance, routage inter-domaine, BGP-4, TCP/IP.

Abstract

This thesis presents a performance evaluation of an implementation of the inter-domain routing protocol BGP-4 (Border Gateway Protocol). This evaluation will conduct to an improvement of the original implementation. In general, to improve the performance of an implementation is always interesting. However, the improvement of an implementation of this protocol is all the more interesting that it subsists some discussions about the problems caused by BGP even maliciously renamed Border Glue Protocol.

BGP-4 is the protocol that builds the routing table requisite to the forwarding in the global Internet. This protocol is the *de facto* standard. Meanwhile, it requires several improvements to survive, in the medium term, as the inter-domain routing protocol.

Thus, this thesis is based on the BGP-4 protocol and one of its implementation, *Zebra*. This thesis proposes to investigate the performances of this implementation. These evaluations lead up to a weak side of this implementation that this thesis try to improve.

Two versions of the improvement will be considered and implemented. These will be tested as the original implementation was. Finally, more tests will be done to have the possibility to see in detail the behaviour of each version of the implementation and then conclude on the real performances of them.

keywords : performance, inter-domain routing, BGP-4, TCP/IP.

Remerciements

Mes remerciements vont en premier lieu au Professeur Olivier Bonaventure sans qui je n'aurais pas pu effectuer ce mémoire captivant et pour son savoir transmis par ces cours.

Je voudrais remercier Bruno Quoitin, Louis Swinnen, Cristel Pelsser et Steve Uhlig non seulement pour leurs conseils tant sur le travail en tant que tel que sur la rédaction du mémoire mais aussi pour leur bonne humeur en toute cirsconstance.

Finalement, je remercie mes parents et amis qui m'ont supporté et encouragé depuis de nombreuses années déjà... et plus particulièrement celle qui a été présente dans les moments les plus difficiles et sans qui je n'aurais même pas entamé ces études.

Table des matières

Table des figures

Acronyme

ARP	Address Resolution Protocol
AS	Autonomous System
ATM	Asynchronous Transfer Mode
ATMARP	Asynchronous Transfer Mode Address Resolution Protocol
ARPANet	Advanced Research Project Agency Network
BGP	Border Gateway Protocol
BGPd	Border Gateway Protocol daemon
DARPA	Defense Advanced Research Projects Agency
DoD	Department of Defense
DP	Decision Process
FIFO	First In First Out
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol (version 1 ou 2)
IGP	Interior Gateway Protocol
IP	Internet Protocol (version 4 ou 6)
ISP	Internet Service Provider
LOC-RIB	Local Route Information Base
MAC	Media Access Control
MED	Multi-Exit-Discriminator
MPLS	MultiProtocol Label Switching
MRT	Multi-threaded Rooting Toolkit
OSPF	Open Shortest Path First (version 2 ou 3)
PIB	Policy Information Base
RIB	Routing Information Base
RIB-IN	Routing Information Base in
RIB-OUT	Routing Information Base out
RIP	Routing Information Protocol
RIP-II	Routing Information Protocol II
RIPng	Routing Information Protocol next generation
RR	Route Reflector
RR-client	Route Reflector-client
SSLD	sender side loop detection
TCP	Transport Control Protocol
VPN	Virtual Private Network

Introduction

Description de l'Internet

En 1969, le DARPA[1] met en place les premiers réseaux interconnectés qui comptaient 4 noeuds, ce réseau se dénommait ARPANet. Fin des années 70, l'architecture et les protocoles avaient déjà acquis la forme actuelle. DARPA était le premier centre de recherche sur les réseaux à commutation de paquets. Il était dépendant du DoD[2] qui en commanditait les recherches. D'ailleurs, ARPANet a d'abord été conçu pour pouvoir survivre à une attaque nucléaire. Ce réseau a pendant longtemps interconnecté chercheurs et institutions militaires. Début des années 80, le réseau passe complètement à TCP/IP.

On compare souvent l'architecture de TCP/IP à l'architecture OSI, pourtant celle-ci n'a pas vraiment été pensée. En effet, ce n'était pas la préoccupation des créateurs de ce réseau à cette époque-là. C'est pour cela que l'architecture TCP/IP diffère en de nombreux points de l'architecture OSI. L'architecture de TCP/IP au sens strict ne comporte que deux couches. La couche transport de la couche OSI représentée par le protocole TCP et la couche réseau représentée par IP. Cependant il existe encore certaines interactions entre celles-ci que OSI ne conseille évidemment pas. Par extension, on considère maintenant que l'architecture TCP/IP comporte 4 couches. Tandis que le modèle OSI en comporte 7.

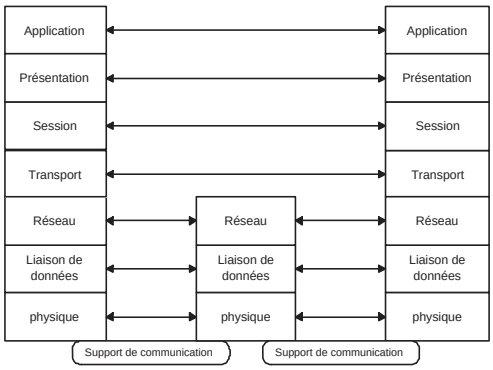


FIG. 1 – *Architecture OSI*

Présentation des couches OSI Les 7 couches du modèle OSI sont représentées sur la figure 1. Chacune de ce couches vont être succinctement décrites dans les paragraphes suivants.

couche physique
La couche physique se charge de transférer les données bit par bit sur la ligne physique. Pour ce faire, elle code les informations pour que la destination puisse décoder le signal émis sans ambiguïté.

couche liaison de données
La couche liaison de données s'occupe de transférer une trame entre deux hôtes du réseau. Elle peut s'occuper de l'ouverture d'une connexion, du transfert de données et de la fermeture de cette connexion. Elle a aussi pour but de détecter/corriger les erreurs survenues lors de l'émission et de réguler le flux entre ces deux hôtes.

[1] Defense Advanced Research Projects Agency.
[2] Department of Defense.

couche réseau

La couche réseau permet de connecter plusieurs hôtes entre eux. Elle s'occupe du routage et du forwarding des paquets dans un réseau ce qui revient à dire que c'est à elle qu'incombe la responsabilité de l'acheminement d'un paquet jusqu'à sa destination finale. C'est grâce à cette couche que divers types de réseaux vont pouvoir cohabiter. Et enfin, elle s'occupe aussi de la fragmentation/défragmentation des paquets et du réordonnancement des datagrammes lorsque c'est nécessaire. La fragmentation consiste à couper les paquets en plus petits paquets lorsqu'un support physique ne supporte pas l'envoi d'un paquet au-dessus d'une certaine taille. La défragmentation est l'opération inverse. Le réordonnancement comme le terme l'indique remet les datagrammes dans l'ordre dans lequel ils ont été envoyés.

couche transport

La couche transport est la première des couches à établir une connexion directe entre l'émetteur et le récepteur des données. Elle permet de savoir, lors de la réception de données, à quelle application ces données appartiennent. Il s'agit donc d'une communication entre deux applications. Elle va permettre de définir le type de service qu'elle doit fournir à l'utilisateur ainsi qu'un contrôle sur le flux.

couche session

La couche session permet de gérer un dialogue qu'il soit bi- ou unidirectionnel. Pour ce faire, elle dispose d'un jeton permettant de contrôler qui a la parole à un moment donné dans le cas d'une conversation unidirectionnelle. Elle permet aussi de faire des points de sauvegarde dans le flot de données au cas où une panne interviendrait. Grâce à ces points de sauvegarde, le flot de données ne doit pas être entièrement rejoué lors du prochain transfert de données.

couche présentation

La couche présentation permet de communiquer avec des machines n'ayant pas les mêmes représentations de données. Elle permet donc de définir des types abstraits que les deux machines utiliseront au long de leur communication.

couche application

La couche application définit des protocoles de haut niveau. C'est via celle-ci que deux machines n'ayant rien en commun vont quand même pouvoir utiliser entre elles des applications effectuant les mêmes opérations.

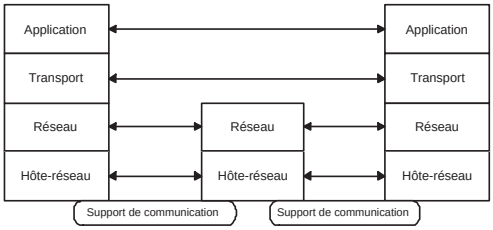


FIG. 2 – *Architecure Internet*

Présentation des couches de TCP/IP De même que pour le modèle OSI, les 4 couches qui sont représentées sur la figure 2 vont être décrites dans les paragraphes suivants.

Couche hôte-réseau

Le modèle TCP/IP ne présente quasiment rien pour la couche hôte-réseau. Il indique simplement que c'est dans cette couche que l'on doit retrouver un protocole permettant d'envoyer des paquets de la couche réseau. A l'heure actuelle, ces protocoles sont bien évidemment existants et scindés de telle manière que l'on puisse retrouver la couche physique et liaison de données du modèle OSI.

Couche réseau

La couche réseau s'occupe de diriger tout datagramme vers sa destination dans des réseaux hétérogènes. On verra dans la prochaine section comment IP se débrouille pour acheminer ces datagrammes jusqu'à destination. Ces datagrammes pouvant arriver dans le désordre, elle s'occupe aussi de les réordonner. IP, le protocole de cette couche, peut aussi fragementer/défragmenter les datagrammes qu'elle envoie. Cette couche correspond aussi à la couche réseau du modèle OSI.

Couche transport

Dans la couche transport de TCP/IP, il existe deux protocoles radicalement différents :

- **Transmission Control Protocol (TCP)** : Ce protocole est basé sur un service orienté connexion et fiable. Le service orienté connexion consiste à établir une connexion avec l'hôte distant avant tout envoi de données. Le service fiable, quant à lui, garantit que les paquets envoyés arrivent bien à destination. TCP s'occupe de transmettre les données entre deux hôtes sans erreur et en contrôlant le flux entre ces deux machines de telle sorte que le flux de données de l'émetteur ne submerge pas un récepteur plus lent.
- **User Datagram Protocol (UDP)** : UDP est quant à lui un protocole non fiable et orienté sans connexion. Contrairement à TCP, il n'existe aucun mécanisme de garantie de transmission entre deux hôtes. De même il n'y a aucun contrôle de flux non plus.

Couche application

De même que pour la couche OSI, c'est dans la couche application que se trouve tous les protocoles de haut niveau comme telnet, ftp, ...

Routage et forwarding dans l'Internet

Dans cette architecture, il a fallu pouvoir atteindre n'importe quel hôte distant. Pour pouvoir atteindre un hôte distant, il faut d'abord connaître l'adresse de l'hôte distant, qui est représentée par l'adresse IP. Ensuite, le destinataire doit pouvoir déterminer pour quelle application un paquet est destiné, ceci s'effectue par le numéro de port. Finalement, il faut connaître aussi l'identifiant de la couche liaison de données, qui est représentée par l'adresse de liaison de données, appelée adresse MAC. Notons que l'adresse MAC n'est pas forcément l'adresse du destinataire. En effet, la couche liaison de données de l'émetteur n'est en communication directe avec le destinataire que s'il est sur le même sous-réseau que l'émetteur.

Une fois ces données connues, on a besoin :

- du routage, qui est l'algorithme par lequel les informations nécessaires vont être distribuées entre les routeurs pour pouvoir savoir par où un paquet doit être envoyé pour joindre sa destination,

– et du forwarding, qui est la décision du chemin que le datagramme réceptionné doit suivre pour atteindre sa destination.

Lorsque l'on essaye de joindre un hôte distant, plusieurs cas peuvent se présenter :

1. *Les deux hôtes sont sur le même sous-réseau :*

   Lorsque l'on cherche à atteindre un hôte distant, on dispose en général de l'adresse IP et du port distant. Cependant, on ne dispose généralement pas de l'adresse MAC. Un hôte peut déterminer s'il est sur le même réseau local que l'hôte de destination grâce à son masque. Dans un premier temps, une requête ARP[Tou99] est diffusée sur le réseau local avec l'adresse IP de l'hôte distant et une adresse MAC de diffusion (void figure 3). Lorsque cet hôte reconnaît son adresse IP, il renvoie une réponse à l'émetteur de ce message en y indiquant son adresse MAC[3].

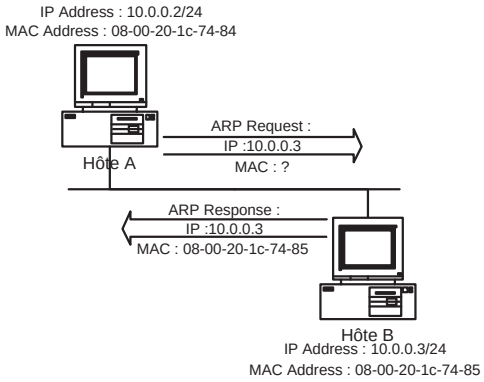


FIG. 3 – *Illustration du protocole ARP*

   Le mécanisme expliqué ci-dessus n'est pas à proprement parler du routage. Par contre, il est indispensable pour pouvoir effectuer le forwarding par la suite.

2. *Les deux hôtes ne se trouvent pas sur le même sous-réseau mais bien dans le réseau d'une même administration :*

   Lorsque l'on parle de réseau sous la même administration, on fait référence au réseau local d'une entreprise.

   Pour ce cas, il faut un routeur entre les deux machines, et c'est par ce routeur que les paquets vont transiter. Un routeur est une machine étant connectée à au moins deux sous-réseaux différents et étant capable de diriger les paquets vers la bonne destination. Celui-ci a, pour cet effet, une table de routage. Une table de routage est constituée d'adresses IP mises en correspondance avec une interface de sortie. Une interface de sortie est une interface réseau appartenant au routeur par laquelle un datagramme doit partir pour atteindre une destination. La table de routage est construite grâce à un protocole de routage interne (comme RIP, ou OSPF).

   Sur la figure 4, lorsque l'hôte B veut envoyer un paquet à l'hôte A, celui-ci sait que l'hôte A ne se trouve pas sur son sous-réseau grâce au masque d'adresse IP. Alors,

[3]Il est fait mention du cas le plus fréquent où l'on se trouve sur un réseau Ethernet mais les mécanismes sont assez similaires pour d'autres réseaux. Par exemple, pour le réseau ATM[Ibe97], c'est un peu plus complexe mais finalement assez ressemblant. Il existe un serveur ATMARP auprès duquel il faut premièrement s'enregistrer, avant de pouvoir envoyer n'importe quelle cellule. Une fois enregistré auprès du serveur ATMARP, on peut effectuer une requête ATMARP auprès de ce serveur pour connaître l'adresse ATM de l'hôte distant avec lequel on veut communiquer.

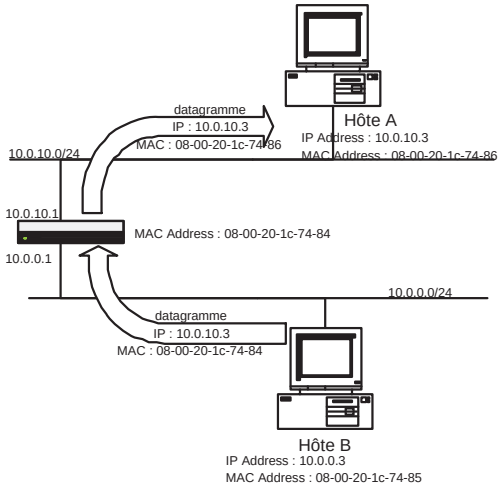


FIG. 4 – *Forwarding simple*

l'hôte B envoie ce paquet à l'adresse d'un routeur ou à une adresse par défaut. Il doit donc connaître l'adresse MAC de ce routeur. La première fois qu'il envoie un datagramme vers l'hôte B, il utilise le protocole ARP pour la trouver. Le datagramme est ensuite envoyé avec l'adresse IP de la destination finale et l'adresse MAC du routeur à atteindre. Une fois qu'un routeur réceptionne le message, il vérifie d'abord si le datagramme lui est destiné, s'il ne l'est pas, alors il trouve la correspondance la plus spécifique avec l'adresse IP du datagramme et les adresses IP de sa table de routage. Une fois qu'il a trouvé cette correspondance il fait suivre ce paquet.

Les protocoles de routage interne sont des protocoles pour lesquels l'efficacité du forwarding est demandée. Ces protocoles essayent de distribuer à chaque routeur l'information la plus perspicace possible et dans les délais les plus courts pour être le plus efficient lors du forwarding des datagrammes. Pour ce faire, les protocoles de routage interne construisent une topologie exacte d'un réseau et peuvent se baser sur cette topologie et certaines métriques [4] associées pour pouvoir choisir quel est le meilleur chemin à suivre pour pouvoir atteindre la destination demandée. Ces protocoles sont bien adaptés au forwarding des datagrammes qui doivent arriver à un hôte de l'entreprise.

Sur la figure 5, une deuxième forme de routage apparaît. Dans le cas présenté, l'hôte B veut envoyer un datagramme vers l'hôte A. Cependant, l'hôte B est configuré pour envoyer ses datagrammes vers une adresse par défaut, qui se trouve être le routeur Rb. Une fois que Rb reçoit ce datagramme, il détermine l'interface de sortie pour faire suivre le datagramme et l'adresse du prochain saut. L'adresse de prochain saut est l'adresse du routeur Ra. Tandis que l'interface de sortie est la même que l'interface par laquelle ce datagramme est arrivé. Alors, le routeur Rb, après avoir forwardé le datagramme envoie un message ICMP redirect vers l'hôte B pour lui indiquer qu'il existe un meilleur chemin pour atteindre l'hôte A en passant par le routeur Ra.

[4] Ces métriques sont encore déterminées par les ingénieurs réseaux pour avoir le forwarding le plus efficace.

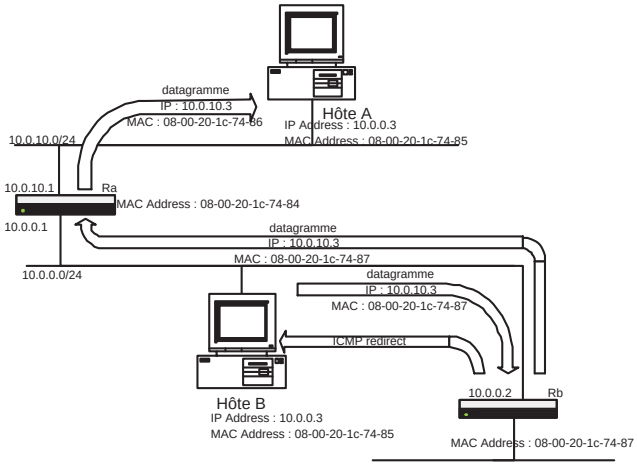


FIG. 5 – *Redirection ICMP*

3. *Les deux hôtes ne se trouvent pas sur le même sous-réseau et pas non plus dans le réseau d'une même administration :*

   Lorsque l'on fait référence à des réseaux n'étant pas sous la même administration, on parle de plusieurs réseaux chacun d'entre eux appartenant à une entreprise différente. L'interconnexion de tous ces réseaux est ce qui constitue l'Internet actuel.

   Lorsque le datagramme doit passer dans le réseau d'une autre administration, il faut qu'il passe par un routeur joignant ces deux réseaux. Ce routeur qui fait tourner un protocole de routage inter-domaine possède une table ayant des adresses IP mises en correspondance avec une adresse de prochain saut. Seulement, cette adresse peut aussi ne pas être sur le même sous-réseau. Il s'agit alors de revenir au cas précédent en cherchant une entrée par rapport à l'adresse de prochain saut.

   Les protocoles s'occupant de construire les tables de routage inter-entreprises sont appelés protocoles de routage inter-domaine (ou externe). Il existe trois grandes différences par rapport aux protocoles de routage interne comme OSPF ou RIP :

   (a) Les entreprises, par lesquelles les datagrammes transitent, ne souhaitent pas diffuser la topologie exacte de leur réseau aux autres entreprises.

   (b) Il serait impossible de diffuser les adresses IP comme dans les protocoles de routage interne. Les tables de routage seraient beaucoup trop grandes. Cela prendrait non seulement trop de place en ce qui concerne la mémoire des routeurs mais aussi trop de temps pour effectuer la correspondance la plus spécifique lors du forwarding d'un paquet.

   (c) Comme l'Internet n'est finalement qu'une interconnexion inter-entreprises de réseaux, celles-ci se font rémunérer pour accepter de faire office de lieu de passage d'une partie du trafic de l'Internet qui ne leur est pas destiné. Cela veut dire que dans les protocoles de routage, il n'est plus question d'avoir un forwarding des datagrammes qui soit le plus performant possible mais bien de le faire de manière à ce que le coût soit le moins élevé possible.

Sujet du mémoire

Le sujet de ce mémoire repose sur le routage inter-domaine dans l'Internet. Plus particulièrement, il traite du protocole BGP version 4. Le but du travail est d'effectuer d'une part une analyse des performances d'une implémentation du protocole, *Zebra*. D'autre part, à partir de cette analyse, il s'agit de déterminer un point faible de cette implémentation et d'essayer d'améliorer l'implémentation étudiée.

Pour ce faire, dans le chapitre premier, le protocole BGP-4 est présenté de manière assez large. Celui-ci approfondit la notion de protocole de routage inter-domaine et montre les mécanismes utilisés pour pouvoir rendre viable un protocole de routage au niveau de l'inter-domaine dans l'Internet.

Le chapitre 2 présente brièvement l'architecture de l'implémentation étudiée, *Zebra*. Les mécanismes généraux de traitement y sont exposés pour pouvoir se faire une idée de la manière adoptée par *Zebra* pour appliquer le protocole BGP-4.

L'outil créé pour effectuer l'analyse des performances de ce logiciel est présenté dans le chapitre 3, ainsi que la présentation des séries de tests et des résultats obtenus pour l'implémentation originale de *Zebra*. Ce chapitre met le doigt sur un point faible de l'implémentation.

Le chapitre 4 présente ce qui peut être fait pour améliorer le point faible mis en évidence dans le chapitre précédent. Deux implémentations différentes sont présentées et testées de la même manière que pour les tests effectués dans le chapitre 3 sur l'implémentation non modifiée. Une fois cette série de tests effectués, une autre batterie est effectuée pour pouvoir se faire une idée plus exacte du comportement des deux versions de la modification apportée à *Zebra*.

1 Le protocole de routage inter-domaine BGP-4

Comme il a été vu dans le chapitre introductif, un protocole de routage inter-domaine est un protocole qui distribue les informations de connectivité entre les différents réseaux de l'Internet n'étant pas soumis à une même administration. Ce chapitre explore plus en profondeur les caractéristiques d'un protocole de routage inter-domaine de l'Internet [5] et les mécanismes qui permettent sa mise en oeuvre. Tout est axé sur le protocole de routage BGP-4 puisque c'est sur celui-ci que le travail se base.

1.1 Premier aperçu du fonctionnement de BGP-4

Le protocole de routage inter-domaine BGP-4 est composé de trois phases :

1. ***ouverture de session*** : Pendant cette phase, les routeurs initient une connexion TCP entre eux et peuvent négocier certaines options de comportement du protocole. Lorsque la session est initiée, les deux routeurs doivent s'échanger les routes actives qu'ils acceptent d'annoncer à l'autre pair.
2. ***échange d'informations de routage*** : C'est la phase active de BGP-4. Pendant cette phase chaque routeur maintient une base de données constituée des routes que le routeur utilise pour effectuer le forwarding des datagrammes. Cette base de données est nommée LOC-RIB [6]. Chaque route de la LOC-RIB doit être annoncée aux pairs avec lesquels il a initié une session. A chaque réception d'une annonce, le routeur doit confronter les routes apprises par cette annonce aux routes faisant partie de la base de données des routes actives. Chaque route est composée d'un préfixe IP et d'attributs - ou, d'une manière générale, caractéristiques - décrivant la route. BGP doit choisir pour chaque route la meilleure des routes.

   La constitution de la base de données de départ de BGP-4 peut être effectuée grâce à deux mécanismes :

   (a) Une première solution consiste à configurer statiquement les routes que BGP doit annoncer.

   (b) La deuxième solution consiste à injecter des routes dans BGP par l'intermédiaire du protocole de routage interne [7].

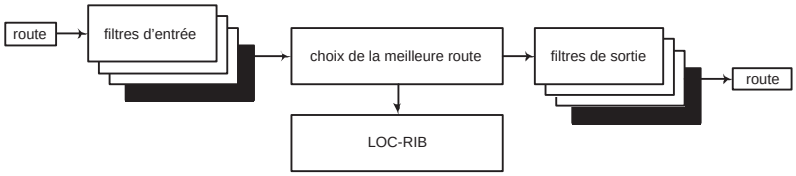


FIG. 1.1 – *Traitement d'une annonce par BGP-4*

Lorsque BGP apprend une route, il n'est pas obligé d'accepter de prendre en compte cette route. De même, lorsque BGP a trouvé le meilleur chemin vers une destination, il n'est pas obligé de l'annoncer vers les autres pairs avec lesquels il communique. Ce comportement est obligatoire dans un protocole de routage externe car il faut pouvoir choisir ce que l'on veut bien accepter recevoir dans son réseau. L'annonce d'une route

[5] En fait, BGP-4 est le seul protocole de routage inter-domaine subsistant à l'heure actuelle dans l'Internet mais il existe d'autres versions de BGP et surtout son ancêtre EGP.

[6] Local Routing Information Base.

[7] Voir le document [Var93b] pour se faire une idée des relations entre BGP et OSPF.

implique automatiquement que l'on sait joindre une destination en passant par son réseau.

Ce comportement peut se voir sur la figure 1.1, une route arrivant est confrontée à un premier filtre, qui décide de son acceptation. De plus, ce filtre peut aussi accepter l'annonce de cette route tout en changeant certaines de ces caractéristiques. L'acceptation de la redistribution de l'annonce d'une route suit le même mécanisme mais en aval du processus déterminant les meilleures routes. De même, ce filtre en sortie peut interdire la redistribution de l'annonce d'une route mais aussi l'accepter tout en changeant certaines de ces caractéristiques.

Cette politique de routage consiste en une configuration manuelle de chaque routeur permettant de décrire certaines actions à effectuer quand l'on rencontre certaines conditions. Par exemple, il est tout à fait possible de configurer un routeur BGP pour qu'il préfère les routes annoncées par un certain AS plutôt que par un autre. Pour que BGP puisse effectuer toutes ces opérations, il a fallu le doter de plusieurs RIBs. En effet, comme on peut appliquer des filtres définis dans le policy information base (PIB) en fonction d'un certain pair BGP, il faut pouvoir différencier les routes annoncées par ce pair ou les routes à annoncer à ce pair. L'Adj-RIB-In est le RIB qui sauvegarde les informations à propos des annonces reçues de la part d'un pair BGP, et l'Adj-RIB-Out est la structure gardant l'information sur les routes à annoncer à un pair [8].

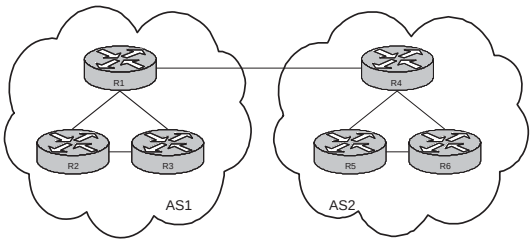


FIG. 1.2 – *Topologie simple de routeurs BGP-4*

En général, plusieurs routeurs BGP appartiennent à un AS. Chacun d'entre eux peut partager des informations avec des AS externes. Sur la figure 1.2, on aperçoit deux AS, l'AS1 et l'AS2, chacun d'eux partage sa connaissance de connectivité avec l'autre. Lorsque le routeur R1 apprend une route de la part d'un autre routeur BGP avec lequel il est connecté, il apprend la connectivité que l'AS veut bien partager avec l'AS1. Pour que l'entièreté du réseau de l'AS1 puisse connaître cette annonce, il faut que le routeur R1 distribue cette information à tous les autres routeurs BGP de l'AS1. On peut donc remarquer qu'il existe deux relations différentes selon que l'on distribue la connectivité à l'extérieur de l'AS ou à l'intérieur. Cette relation s'appelle session eBGP ou iBGP selon que, respectivement, une session est initiée avec un routeur BGP à l'extérieur de l'AS ou à l'intérieur de l'AS [9].

3. ***fermeture de session*** : Cette phase n'est pas explicitement demandée par un des pairs mais peut arriver lorsqu'il y a une erreur qui survient. Entendons par erreur, tout ce qui n'est pas prévu dans le déroulement normal du protocole (exemples :

[8] Notons que le protocole n'impose aucune contrainte structurelle d'implémentation sur les RIBs. Le protocole ne définit ces notions que d'une manière conceptuelle.

[9] iBGP impose certaines contraintes expliquées en section 1.5 page 23.

rupture de session TCP, erreur syntaxique d'un message, ...). Lors d'une terminaison de session avec un pair, toutes les routes actives ayant été annoncées par celui-ci doivent être retirées de la base de données des routes actives. Et il faut à nouveau choisir de meilleures routes parmi celles qui ont été annoncées par les autres pairs avec lesquels il existe une session BGP. Pouvoir choisir à posteriori des routes annoncées implique que BGP doive sauvegarder les routes annoncées par ces pairs BGP. Pour ce faire, il suffit d'utiliser la même structure que celle utilisée pour les filtres, l'Adj-RIB-in.

1.1.1 BGP-4, un *path-vector protocol*

Le protocole BGP-4 est basé sur un type de protocole de routage connu, le protocole à vecteurs de distance. Cependant, ce type de protocole a besoin d'être modifié pour pouvoir être appliqué au niveau de l'inter-domaine dans l'Internet. Le prochain paragraphe présente donc les caractéristiques d'un protocole à vecteurs de distance pour ensuite montrer quelles sont les différences avec le path-vector protocol.

protocole à vecteurs de distance Un routeur démarrant, et faisant tourner un protocole de routage à vecteurs de distance, initialise sa table de routage avec les liens auxquels il est directement connecté. Une fois cette table initialisée, il doit la distribuer aux autres routeurs auxquels il est connecté. Chaque routeur interceptant un de ces messages peut remplir sa table de routage en n'oubliant pas d'incrémenter de un la distance à laquelle se trouve chaque route annoncée. A chaque fois qu'il y a un changement de la table de routage ou à l'expiration d'un timer, celle-ci est à nouveau redistribuée.

Certains problèmes peuvent apparaître avec ce type de protocole de routage. Ainsi, si, sur la figure 1.3, le lien entre le routeur C et le routeur B s'interrompt, alors le routeur B ne sait plus comment accéder au réseau 192.136/16. Si le routeur A envoie sa table de routage au routeur B, le routeur B sait à nouveau comment accéder au réseau 192.136/16. Le routeur A sachant qu'il a reçu l'annonce de cette route par B remplace cette entrée par la nouvelle annonce faite par B et ainsi de suite. Ce problème est mieux connu sous le nom du **comptage à l'infini**. Un moyen de résoudre en partie ce problème est d'interdire l'annonce d'une route sur le lien par lequel on a appris cette route, ce qui s'appelle l'**horizon partagé**. Cependant cette manière de faire ne résout pas tous les problèmes.

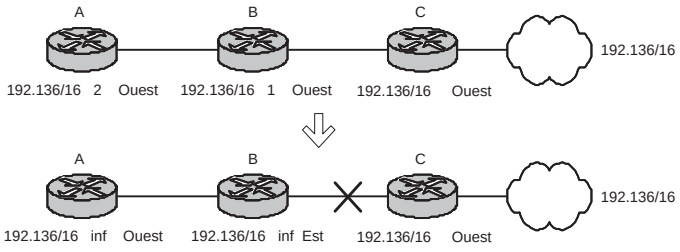


FIG. 1.3 – *Comptage à l'infini*

Il existe d'autres solutions pour éviter ce comptage à l'infini comme l'horizon partagé avec empoisonnement. BGP quant à lui utilise un autre moyen de résoudre le comptage à l'infini. Il distribue le chemin complet par lequel devrait passer un datagramme et n'accepte jamais une route s'il se trouve déjà dans le chemin annoncé par cette route. On appelle,

par extension, ce type de protocole un **path vector protocol**.

BGP-4, un path vector protocol BGP n'est donc pas un simple protocole de routage à vecteurs de distance mais un protocole à vecteurs de distance modifié. Un protocole de routage à vecteurs de distance n'est viable que pour un petit réseau local parce qu'il distribue à chaque fois toute sa table de routage. Il est en effet inimaginable d'annoncer toutes les routes possibles à atteindre au niveau de l'Internet global[10] à chaque fois qu'un changement survient dans le réseau ou à chaque expiration d'un timer.

Il a donc fallu améliorer la manière d'annoncer les routes possibles pour pouvoir baser BGP sur ce type de protocole de routage. L'amélioration apportée pour que le protocole BGP soit viable au niveau de l'inter-domaine a été dans un premier temps de considérer que toute route annoncée reste valable indéfiniment à moins que l'on annule explicitement cette route. Ce changement induit deux changements majeurs dans la manière de propager les informations de routage. Premièrement, il a fallu ajouter un moyen d'annuler des routes annoncées et ceci peut se faire de deux manières[11] :

- soit en annulant explicitement une route via un message bien défini,
- soit en prenant le cas où un routeur, ayant déjà annoncé une route, l'annonce à nouveau parce que certaines caractéristiques de cette route ont changé. Ce mécanisme s'appelle une annulation implicite puisque l'on peut considérer que la nouvelle annonce annule l'ancienne annonce.

Deuxièmement, BGP effectue ses annonces incrémentalement. Ceci signifie que lorsqu'il y a un changement dans le réseau, le routeur n'annonce pas à nouveau toutes ses routes actives mais uniquement celles qui ont subi des changements d'états.

Ce mécanisme d'annonce peut épargner énormément de bande passante sur le réseau. Ceci est vrai dans la théorie, dans la pratique, c'est moins évident qu'il n'y paraît, pour des raisons de défaillances matérielles de perte de connectivité, d'implémentations ou autres[12].

1.1.2 BGP-4 et le CIDR

Malgré l'amélioration majeure des annonces incrémentales, il a fallu résoudre un autre problème dû à la popularité grandissante de l'Internet : le concept de classe associé à une adresse IPv4. La croissance exponentielle de l'allocation des adresses IPv4[13] a obligé les routeurs à avoir de plus en plus de capacité mémoire mais aussi de générer de plus en plus de trafic. Ainsi, jusqu'au concept de CIDR (Classless Inter-Domain Routing), chaque adresse était associée à une des 5 classes existantes : A, B, C, D, E. L'existence de 5 classes ne permet pas d'agréger correctement les informations. Ce qui a eu comme conséquence une surcharge des routeurs. Le concept de "Classless Inter-Domain Routing" (CIDR) élimine l'ancien concept de classes tout en gardant le concept d'identification réseau et hôte. Il est toujours possible d'identifier un réseau de l'Internet en indiquant le nombre de bits à prendre en compte pour l'identifier. Donc, en prenant les 24 premiers bits de l'adresse 192.38.10.0/24, nous pouvons identifier le sous-réseau et considérer que les 8 bits restants

[10] Rappelons qu'il existe à peu de choses près 2^{32} adresses possibles pour IPv4 et 2^{128} adresses possibles pour IPv6.

[11] La deuxième manière existait déjà pour le protocole de routage à vecteurs de distance non modifié.

[12] Voir la papier suivant [Wet02] pour une présentation des raisons de l'instabilité dans l'Internet.

[13] Voir la page de Geoff Huston sur la croissance des routes actives dans BGP à l'adresse suivante : http ://bgp.potaroo.net.

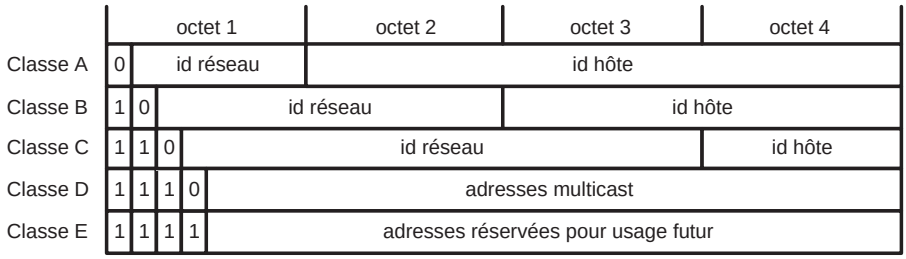


FIG. 1.4 – *Les 5 classes de l'adressage IPv4*

servent à identifier les hôtes de ce réseau. De cette manière, il est possible d'agréger au mieux les informations et à tous les niveaux du routage inter-domaine.

Après avoir vu les changements obligatoires pour que ce protocole puisse être viable au niveau de l'inter-domaine, nous allons voir plus en profondeur tous les mécanismes dont il dispose pour pouvoir effectuer correctement le routage inter-domaine et aussi ses mécanismes qui permettent d'influencer le routage inter-domaine[14].

1.2 Description des messages

Le protocole de base BGP définit 4 messages de base dont une description succincte est donnée dans la suite de cette section pour trois de ceux-ci et une description plus détaillée en ce qui concerne le message UPDATE car c'est celui-ci qui nous intéresse dans la suite de ce travail. D'autres messages ont aussi été définis par des extensions du protocole. Ces extensions sont abordées lors des sections suivantes. En ce qui concerne le format des messages, ils sont tous décrits dans l'annexe B à la page 82.

1.2.1 HEADER

Chaque message décrit par la suite doit commencer par l'en-tête suivant :

Nom du champ	longueur
Marker	16 octets
Length	2 octets
Type	1 octet

FIG. 1.5 – *Format de l'en-tête*

- *Marker* : contient une valeur que celui qui le réceptionne sait prédire. Sert pour l'authentification et pour la détection de désynchronisation entre deux pairs.
- *Length* : la valeur de ce champ représente la longueur totale du message (header inclus). Le paquet doit obligatoirement avoir une longueur au moins égale à 19 octets et au plus à 4096.
- *Type* : la valeur de ce champ définit le type du message.
  1. OPEN

[14] La description du protocole BGP-4 qui va suivre se base principalement sur les deux documents suivants [Rek02d] et [III99].

2. UPDATE
3. NOTIFICATION
4. KEEPALIVE

1.2.2 OPEN message

Le message OPEN est le premier message envoyé à un autre pair BGP lorsque l'on veut initier une session avec lui. Lorsqu'un pair reçoit un message OPEN valide et qu'il est en droit de le recevoir au vu de l'état courant de la machine à état, alors il doit renvoyer un message KEEPALIVE en confirmation de la réception de ce message.

Il est possible via ce message d'annoncer certaines capacités qu'un routeur voudrait utiliser et qui ne sont pas obligatoirement implémentées[15]. Dans le draft décrivant le protocole BGP, il n'y a qu'une option définie. Cette option a pour but de négocier l'utilisation du mécanisme d'authentification [Hef02].

Il existe un défaut majeur à ce mécanisme de négociation de capacités ; il faut absolument couper la session entre les deux pairs si l'on veut rajouter une capacité ou en enlever une. Donc, pour améliorer cette fonctionnalité, un message a été rajouté. Celui-ci permet de négocier dynamiquement les capacités de BGP-4 [San02]. La création de ce message n'a toutefois pas enlevé à BGP-4 la capacité de négocier certaines options avec le message OPEN.

1.2.3 UPDATE message

C'est par le message UPDATE de BGP que l'on va pouvoir annoncer plusieurs routes que l'on peut joindre et partageant les mêmes attributs et/ou annuler certaines de ces routes (ce qui revient à dire qu'elles ne sont plus accessibles à partir du pair qui les avait annoncées). Si on doit annoncer plusieurs routes joignables mais qu'elles n'ont pas les mêmes attributs, il faut alors les annoncer par plusieurs de ces messages UPDATE.

format du message UPDATE Il ne faut pas oublier qu'il y a toujours le préambule à ajouter au format décrit ci-dessous (voir figure 1.5).

Nom du champ	longueur
Withdrawn Routes Length	2 octets
Withdrawn Routes	variable
Total Path Attribute Length	2 octets
Path Attributes	variable
Network Layer Reachability Information (NLRI)	variable

FIG. 1.6 – *Format du message UPDATE*

- *Withdrawn Routes Length* : il s'agit de la longueur du champ contenant les routes à annuler.
- *Withdrawn Routes* : il s'agit des routes à annuler selon le format suivant :

[15] Voir le RFC 2842 [Scu02] pour plus d'informations sur cette capacité de négocier des options avec le message OPEN.

Nom du champ	longueur
Length	1 octet
Prefix	variable

FIG. 1.7 – *Format d'un préfixe*

- *length* : longueur du préfixe en nombre de bits.
- *prefix* : valeur d'une adresse IP. Cette valeur doit être finie par des zéros pour obtenir un alignement sur un octet.

Ce champ a une longueur égale à la valeur du champ *Withdrawn Routes Length*.

- *Total Path Attributes Length* : il s'agit de la longueur du champ *Path Attributes*.
- *Path Attributes* : Ce champ définit tous les attributs concernant les préfixes annoncés dans le *NLRI*. Ces attributs sont les éléments importants, avec le préfixe, du fonctionnement de BGP. C'est entre autres grâce à une partie de ceux-ci que l'on va pouvoir déterminer quel va être la meilleure route.
  Tous les attributs ont le format de la table 1.8 :

Nom du champ	longueur
Attribute Type	2 octets
Attribute Length	1 or 2 octets
Attribute Value	variable

FIG. 1.8 – *Format des attributs*

- *Attribute Type* : Il s'agit d'un champ sur deux octets. Le premier octet définit l'*Attribute Flags* et le second définit l'*Attribute Type Code*. Comme le montre la figure 1.9, les 4 bits de poids fort de l'*Attribute Flags* ont une signification tandis que les autres doivent obligatoirement être mis à 0 car ils sont inutilisés.

Optional	Transitive	Partial	Extended Length	0000

FIG. 1.9 – *Format de l'Attribute Flags*

  - *Optional* bit : Ce bit détermine si l'attribut est "bien connu" (valeur à 0) ou optionnel (valeur à 1). Si l'attribut est "bien connu" alors toutes les implémentations doivent savoir le traiter. Tandis que s'il est optionnel, il n'est pas obligatoire qu'une implémentation le reconnaisse.
  - *Transitive* bit : Pour les attributs "bien connu", ce bit doit être mis à 1. Cela signifie que lorsqu'une route contient un attribut transitif, celui-ci fait obligatoirement partie de cette route quand celle-ci est redistribuée à d'autres pairs.
  - *Partial* bit : Ce bit est mis à 1 lorsqu'un routeur BGP reçoit un préfixe avec un attribut optionnel transitif et qu'il ne sait pas traiter l'attribut. Comme cet attribut est transitif, il doit faire suivre le préfixe vers un autre pair avec cet attribut dans le *Path Attribute*. Ainsi, l'autre pair peut déterminer qu'il y a eu une perte d'information dans cette annonce. Il est mis obligatoirement à 0 si l'attribut est "bien connu" ou optionnel et non transitif.
  - *Extended Length* bit : Si le bit est à 0 alors l'*Attribute Length* est d'une longueur d'un octet. Si ce bit est à 1 alors l'*Attribute Length* est de deux octets.

  L'*Attribute Type Code* est un nombre assigné par l'IANA[16] qui identifie l'attribut.

[16] Internet Assigned Numbers Authority.

- *Attribute Length* : la valeur de ce champ représente la longueur de l'*Attribute Value*.
- *Attribute Value* : Au vu de l'importance des attributs et de leur nombre, ceux-ci sont décrits par la suite. Les attributs de base sont décrits dans la section 1.3 tandis que les attributs créés lors d'extensions à BGP sont décrits lors de la présentation même de l'extension.

– NLRI : Ce champ est prévu pour contenir une liste de préfixes d'adresses IP et est encodé de la manière suivante :

Nom du champ	longueur
Length	1 octet
Prefix	variable

FIG. 1.10 – *Format du NLRI*

Le champ *Length* détermine la longueur du champ *Prefix*. La valeur de ce champ doit être comprise entre 0 et 4. Le champ *Prefix* contient un préfixe d'une adresse IP.
La longueur totale du NLRI peut être calculée selon la formule suivante :
UPDATE Message Length−23−*Total Path Attributes Length*− *Withdrawn Routes Length*.

Le message UPDATE peut annoncer plusieurs préfixes dans le même message, seulement si ceux-ci ont exactement les mêmes attributs. Ce qui revient à dire que deux préfixes à annoncer n'ayant pas les mêmes attributs sont annoncés dans des messages UPDATE distincts. De même, un message UPDATE peut contenir plusieurs préfixes à annuler.

1.2.4 NOTIFICATION message

Le message NOTIFICATION indique une erreur de déroulement dans le protocole défini par BGP-4. Lors de l'envoi d'un tel message, BGP rompt la session avec le pair pour lequel il envoie ce message. Lors d'une rupture de session avec un pair, les routes apprises par ce pair sont annulées.

1.2.5 KEEPALIVE message

Le protocole BGP-4 n'utilise pas un moyen de détecter les coupures de liaison basé sur la couche transport. De ce fait, BGP-4 utilise un message qui est envoyé à intervalles réguliers, si aucun message UPDATE n'est envoyé[17], et dont la seule utilité est d'indiquer sa présence à un autre pair ou d'effectivement remarquer qu'il n'existe plus de liaison entre deux pairs. Ce message s'appelle le message KEEPALIVE et est juste composé de l'HEADER décrit précédemment (voir figure 1.5).

1.3 Attributs de base d'un préfixe

Tous les attributs décrits dans cette section sont les attributs de base du protocole BGP [Rek02d]. En ce qui concerne les attributs rajoutés lors d'extensions au protocole, ceux-ci sont décrits lors de la présentation de ces extensions.

[17] Une discussion se déroule actuellement pour savoir s'il faut généraliser ce mécanisme à tout autre message autre que le message OPEN et ne plus être restreint au message UPDATE.

1.3.1 ORIGIN

L'attribut ORIGIN permet de savoir comment un pair a appris une route. En effet, les préfixes sont appris par BGP par l'intermédiaire :
- d'une configuration statique,
- d'interfaces directement connectées,
- de protocoles de routage dynamiques externes,
- de protocoles de routage dynamiques internes.

Les valeurs de cet attribut sont :

1. **IGP** : signifie que ce préfixe a été appris à partir d'un protocole de routage interne.
2. **EGP** : signifie que ce préfixe a été appris à partir d'un protocole de routage externe.
3. **INCOMPLETE** : signifie que le préfixe n'a ni été appris par un IGP ni par un EGP.

Le code du type de cet attribut est 1. Cet attribut doit être présent dans toutes les annonces et il est du type "bien connu" et transitif.

1.3.2 AS-PATH

L'attribut AS-PATH contient la liste de tous les numéros d'AS par lequel l'annonce est déjà passée. Grâce à cet attribut, BGP peut prévenir des bouclages dans les annonces. Il est composé d'une séquence de segments d'AS-PATH. Chaque segment est composé du triplet représenté par la figure 1.11.

Nom du champ	longueur
Path Segment Type	1 octet
Path Segment Length	1 octet
Path Segment Value	2 octets

FIG. 1.11 – *Format d'un segment d'AS-PATH*

- *Path Segment Type* : la valeur de ce champ peut être soit 1 ou 2 selon que ce soit respectivement un AS-SET ou un AS-SEQUENCE. L'AS-SET contient un ensemble de numéros d'AS désordonnés. Tandis que l'AS-SEQUENCE contient un ensemble de numéros d'AS ordonnés.
  L'existence de l'AS-SET permet de supporter l'agrégation de routes ayant des AS-PATH différents. Imaginons qu'un pair BGP ait deux préfixes 138.48.128/17 et 138.48.0/17 avec, pour l'un, un AS-PATH de "1 2 3" et, pour l'autre, un AS-PATH de "4 2 3". Dans ce cas-ci, le pair BGP peut agréger les deux préfixes et obtenir le préfixe 138.48/16. En ce qui concerne les deux AS-PATH, il va falloir en recréer un autre. Celui-ci est égal à un premier segment de type AS-SET égal à [1, 4] et un autre segment de type AS-SEQUENCE égal à "2 3". Notons que l'agrégation de deux routes ne peut se faire que si les valeurs des champs NEXT-HOP et MED sont les mêmes.
- *Path Segment Length* : la valeur de ce champ détermine la longueur du *Path Segment Value*.
- *Path Segment Value* : chaque numéro d'AS devant être dans le *Path Segment Value* est encodé sur 2 octets.

Chaque routeur doit effectuer une vérification des numéros d'AS présents dans l'AS-PATH pour vérifier que le numéro d'AS du routeur BGP ayant reçu une annonce ne s'y trouve pas[18]. Cette vérification doit se faire lors de la réception d'un message UPDATE mais il est aussi possible d'épargner un peu de bande passante et de temps processeur pour les autres pairs BGP auxquels on peut faire suivre cette information en vérifiant la même chose avant d'envoyer le paquet. La vérification anticipée de boucle s'appelle le "*sender side loop detection*" (SSLD).

Les ISPs utilisent couramment ce champ pour influencer le processus de décision des autres AS en ajoutant plusieurs fois leur numéro d'AS dans l'AS-PATH. Ceci a la particularité de rendre moins bonne la route annoncée.

Le code du type de cet attribut est 2. Il doit être présent dans toutes les annonces et il est du type "bien connu" et transitif.

1.3.3 NEXT-HOP

La valeur de l'attribut NEXT-HOP permet au pair BGP de savoir vers quelle destination doivent être envoyés les datagrammes à forwarder. L'existence de ce champ induit que le routeur par lequel doivent passer les paquets ne doit pas forcément être le routeur BGP via lequel on doit forwarder un datagramme. Plusieurs règles doivent être prises en compte pour le choix d'une valeur du NEXT-HOP lorsqu'un routeur BGP veut annoncer une route à un de ses pairs[19] :

- *si le pair est un pair interne* :
  La valeur du NEXT-HOP ne doit pas être changée.

- *si le pair est un pair externe à un saut du routeur* :
  Il existe alors quatre possibilités de garnir le champ NEXT-HOP :
  1. Si la route a été apprise par un pair interne ou originaire de l'intérieur du réseau, alors la valeur du NEXT-HOP peut être une des interfaces du pair interne ou du routeur interne. L'interface choisie doit partager un sous-réseau commun avec une des interfaces du routeur externe.
  2. Si la route a été apprise par un routeur externe, la valeur du NEXT-HOP peut être une adresse IP d'un routeur adjacent (et connue du NEXT-HOP reçu). L'interface choisie doit partager un sous-réseau commun avec une des interfaces du routeur externe.
  3. Si le routeur externe auquel on annonce une route partage un sous-réseau commun avec une des interfaces du routeur annonçant, alors on peut utiliser cette adresse comme NEXT-HOP[20].
  4. Si aucune des conditions ci-dessus n'est applicable, alors la valeur du NEXT-HOP doit être l'adresse de l'interface que le pair utilise pour établir la session

[18] Dans certains cas, on peut configurer le routeur pour qu'il n'effectue pas cette vérification. Cela peut être souhaité lorsque la session entre les deux pairs s'effectue au moyen d'un lien virtuel qui traverse un autre AS. La communication entre les deux pairs est vue comme une communication directe. Cependant, si ce lien venait à se rompre, il est alors intéressant de pouvoir disposer de ce mécanisme pour continuer à distribuer les informations de routage entre les deux AS distants.

[19] Il existe d'autres règles à prendre en compte dans le cas des réflecteurs de routes ou des confédérations d'AS. Ceux-ci sont présentés dans la suite de ce chapitre sections 1.5.2 et 1.5.3.

[20] Les deux premiers cas sont appelés "third party" NEXT-HOP tandis que le troisième cas est appelé "first party" NEXT-HOP.

avec le routeur externe.

– *si le pair est un pair externe à plus d'un saut du routeur* :
Il existe deux possibilités de garnir le champ NEXT-HOP :

1. La valeur du NEXT-HOP peut rester la même que la valeur de cet attribut lors de la réception de l'annonce.
2. Par défaut, la valeur du NEXT-HOP doit être l'adresse de l'interface que le pair utilise pour établir la session avec le routeur externe.

Nom du champ	longueur
NEXT-HOP Value	4 octets

FIG. 1.12 – *Format du NEXT-HOP*

Le code du type de cet attribut est 3. Il doit être présent dans toutes les annonces et est "bien connu". Cet attribut est transitif.

1.3.4 MULTI-EXIT-DISCRIMINATOR (MED)

L'attribut MED est utilisé lorsque deux AS sont connectés en plusieurs points. Il est alors possible de l'utiliser pour choisir le meilleur lien pour atteindre un préfixe annoncé passant par un des liens reliant les deux AS. Cet attribut est un attribut numérique configuré par un AS en interne et qui est distribué ensuite au deuxième pair pour qu'il puisse choisir lui-même le meilleur lien. Le deuxième pair choisira le MED le plus petit entre deux routes annoncées de même préfixe. En général cela ne se fait sans des accords passés entre les deux AS concernés.

Si l'on s'en réfère à la figure 1.13, on peut influencer le choix d'une ligne choisie par l'AS1 pour forwarder les paquets en direction de l'AS3 ou de l'AS4. Pour ce faire, l'annonce des routes de l'AS3 vers l'AS1 faite par l'AS2 contient un attribut MED. La valeur de celui-ci dépend du lien sur lequel l'annonce est faite. La valeur du MED est plus grande lorsque l'AS2 fait l'annonce sur le lien B. Il en va de même pour les annonces faites par l'AS2 des routes de l'AS4 vers l'AS1. Il suffit d'inclure un MED pour l'annonce qui est plus grand quand cette annonce est faite sur le lien A que lorsque celle-ci est faite sur le lien B. De par cette configuration, lorsqu'un paquet à destination de l'AS3 arrive dans l'AS1, celui-ci le forwarde par le lien A plutôt que par le lien B.

La connectivité par le lien B n'est pas pour autant perdue. En effet, si le lien A venait à tomber en panne alors l'AS1 se servirait du lien B pour forwarder les paquets à destination de l'AS3.
Le code du type de cet attribut est 4. Il est optionnel et non transitif.

1.3.5 LOCAL-PREF

L'attribut LOCAL-PREF permet à un pair BGP de choisir entre plusieurs chemins annoncés pour le même préfixe. Il y a deux avantages en comparaison avec le MED :

1. l'AS ne dépend pas d'un autre AS pour effectuer son choix,

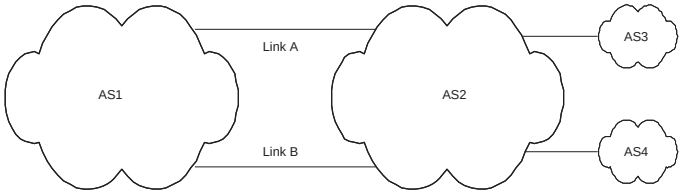


FIG. 1.13 – *Utilisation du MED*

Nom du champ	longueur
MED Value	4 octets

FIG. 1.14 – *Format du MED*

2. on enlève la restriction du MED. Le MED s'utilise entre une **paire** d'AS qui ont au moins deux liaisons directes.

L'AS configure en interne des préférences pour chaque préfixe appris par un autre AS. La route préférée est celle dont la valeur du LOCAL-PREF est la plus élevée. Cet attribut est distribué lors de sessions iBGP mais ne peut pas l'être pour les sessions eBGP. En effet, comme il s'agit d'un attribut influençant la manière dont le trafic va être distribué en interne, ce champ n'a aucun intérêt pour un autre AS.

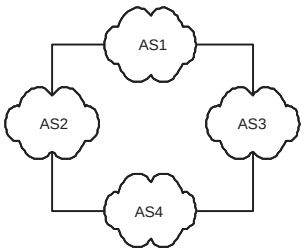


FIG. 1.15 – *Utilisation du LOCAL-PREF*

Ainsi sur la figure 1.15, l'AS1 peut choisir, par exemple, l'AS2 pour son trafic sortant en direction de l'AS4. Pour cela, il suffit qu'il configure localement une valeur du LOCAL-PREF plus grande pour les annonces faites par l'AS2 des routes de l'AS4 que pour celles faites par l'AS3.

Nom du champ	longueur
LOCAL-PREF Value	4 octets

FIG. 1.16 – *Format du LOCAL-PREF*

Le code du type de cet attribut est 5. Il est "bien connu" et non obligatoire.

1.3.6 ATOMIC-AGGREGATE

Imaginons un même AS annonçant deux préfixes dont l'un est le suffixe de l'autre mais qui n'ont pas les mêmes attributs. Le réceptionneur de ces deux préfixes peut n'annoncer

qu'un seul des deux préfixes. Avec l'attribut ATOMIC-ATTRIBUTE, il informe les autres pairs que :

1. ils ne peuvent plus désagréger ce préfixe en préfixes plus spécifiques,
2. ils savent dès lors que les attributs inhérents à ce préfixe ne reflètent pas forcément la réalité. En particulier, l'AS-PATH n'est probablement pas le même que celui annoncé dans chacune des routes agrégées. Par contre, si on considère l'ensemble des numéros d'AS-PATH contenus dans chacune des routes utilisées pour l'agrégation, alors chaque numéro d'AS-PATH distinct se trouve au moins une fois dans l'AS-PATH de la route agrégée.

Le code du type de cet attribut est 6. Il est "bien connu" et non obligatoire. Notez que l'attribut n'a pas de valeur spécifique, il est juste utilisé comme un drapeau. Donc, le simple fait de le définir dans les attributs d'un préfixe suffit à signaler sa présence aux autres pairs.

1.3.7 AGGREGATOR

Si un pair BGP effectue une agrégation, il peut signaler aux autres pairs BGP l'AS qui a fait cette agrégation en spécifiant son numéro d'AS et son adresse IP.

Nom du champ	longueur
AS number	2 octets
IP address	4 octets

FIG. 1.17 – *Format de l'AGGREGATOR*

Le code du type de l'attribut AGGREGATOR est 7. Il est optionnel et transitif.

1.4 Le processus de décision (DP)

Le processus de décision est le processus par lequel BGP va pouvoir choisir les routes préférées pour effectuer le forwarding des paquets. Ces routes préférées sont aussi annoncées, après application des filtres à la sortie, à chaque pair avec lequel une session est en cours.

le processus de décision s'occupe de :
- sélectionner les routes utilisées par le routeur lui-même
- sélectionner les routes à annoncer aux pairs iBGP et eBGP
- effectuer les agrégations de routes et réduction d'information de routes.

Le processus de décision se fait en 3 phases distinctes :
- *Phase 1* : C'est lors de cette phase que le DP calcule le degré de préférence de chaque route apprise par des pairs externes. Cette phase intervient lorsqu'un message UPDATE arrive qui annonce une nouvelle route, un remplacement de route ou une annulation d'une route[21].
  Si la route est apprise par un pair interne, alors il faut soit prendre en compte l'attribut LOCAL_PREF comme degré de préférence soit en calculer aussi le degré de

[21] Il est en effet possible de recevoir un message UPDATE vide avec l'extension "Graceful Restart Mechanism" (voir la section 1.6).

préférence lors de cette phase basé sur une configuration. Si la route est apprise par un pair externe, alors BGP doit en calculer le degré de préférence basé sur une préconfiguration de la politique de routage et utiliser ce degré de préférence comme valeur de LOCAL_PREF dans chaque annonce à un de ses pairs iBGP.

- *Phase 2* : C'est cette étape qui élit la meilleure route parmi toutes celles qui sont disponibles pour chaque destination distincte. Les routes choisies sont aussi installées dans le Loc-RIB.
  Pour chaque ensemble de destination, il faut choisir une route parmi les Adj-RIBs-In qui répond à une des 3 conditions suivantes :
  - c'est la route qui a le plus haut degré de préférence parmi toutes les routes possibles,
  - c'est la seule route existante pour cette destination,
  - c'est la route qui a été sélectionnée en cas d'égalité du degré de préférence entre plusieurs routes par un ensemble de règles visant à les départager.

  L'ensemble des règles visant à départager plusieurs routes qui ont la même destination et le même degré de préférence est décrit ci-dessous. Chaque étape est prévue pour éliminer un certain nombre de routes, s'il y a encore plusieurs routes qui restent à départager alors on passe à l'étape suivante de l'algorithme. Cet algorithme se finit lorsqu'il ne reste plus qu'une route.
  1. Prendre en considération les routes qui ont le plus petit nombre de numéros d'AS dans l'AS-PATH.
  2. Prendre en considération les routes qui ont le plus petit numéro d'origine de l'attribut ORIGIN.
  3. Prendre en considération les routes pour lesquelles le MED est le plus petit. Il est important de noter que le MED n'est comparable que s'il vient du même système autonome. Donc, il est possible d'avoir plusieurs routes restantes après cette étape ayant un MED différent.
  4. Si une des routes encore en considération a été reçue par un pair eBGP, alors il ne faut plus prendre en considération toutes les routes reçues par les pairs iBGP.
  5. Ne plus prendre en considération que les routes pour lesquelles la métrique de l'IGP est la meilleure.
  6. Ne plus prendre en considération que les routes pour lesquelles l'identifiant du pair BGP est le plus petit.
  7. Préférer l'adresse du pair BGP qui a la plus petite adresse.

  Cet algorithme est souvent modifié par les sociétés qui implémentent le protocole. De plus, la plupart du temps, elles laissent la possibilité aux utilisateurs de désactiver certaines des étapes.

- *Phase 3* : Lors de cette phase, BGP dissémine les routes du Loc-RIB à chaque pair BGP en accord avec les politiques contenues dans le PIB. C'est lors de cette phase que l'agrégation de routes et la réduction d'information peuvent être effectuées.
  Cette phase doit être exécutée à chaque fois que l'une des conditions suivantes est rencontrée :
  1. la phase 2 vient de se terminer,
  2. des routes installées dans le LOC-RIB vers des destinations locales viennent de changer,

3. des routes apprises de manière externe à BGP ont changées,
4. une nouvelle session a été initée avec un autre routeur BGP.

1.5 iBGP et eBGP

Les annonces distribuées entre AS doivent aussi l'être à l'intérieur des AS. On parle, respectivement, d'eBGP et d'iBGP. Cela reste le même protocole car ils partagent les mêmes messages mais ils ne suivent pas tout à fait les mêmes règles de distribution de l'information :

- Le premier changement réside dans le fait qu'un routeur BGP apprenant une route par un pair iBGP ne peut redistribuer cette information à un autre pair iBGP. Ceci s'explique parce que le moyen utilisé pour détecter des boucles dans le réseau se base sur les numéros d'AS par lequel l'UPDATE est passé, or il est impossible de détecter une boucle à l'intérieur d'un AS par ce système là. Ce comportement a une lourde conséquence car il impose que tous les pairs iBGP soient interconnectés. Quand il n'y a que deux ou trois routeurs, le nombre de connexions reste raisonnable mais cela ne l'est plus lorsqu'il s'agit d'interconnecter dans un maillage complet une dizaine de routeurs. Il faut en fait établir $N * (N-1)/2$ connexions (N étant le nombre de routeurs iBGP). Cependant, il existe trois approches pour éviter ce maillage complet. Les sous-sections sur les serveurs de routes (1.5.1), la réflexion de routes (1.5.2) et la confédération d'AS (1.5.3) décrivent des solutions possibles à ce problème de maillage complet.
- Le deuxième changement est un problème dû au système d'adressage. Il s'agit d'une astuce de configuration du routeur pour les sessions iBGP. Ces sessions ne peuvent utiliser les adresses liées aux interfaces si l'on veut pouvoir faire fonctionner correctement BGP[22]. Ce point est discuté dans la section 1.5.4.

1.5.1 Serveur de routes

Le principe d'un serveur de routes [Has95] est de placer un routeur qui "centralise" les annonces faites par des border routers et qui les redistribue vers des serveurs de routes clients. Ce serveur de routes n'effectue aucun traitement de préférence sur les routes annoncées. Donc, il n'existe ni filtre à l'entrée, ni filtre à la sortie ni processus de décision. Il existe deux types de serveurs de routes :

1. *les serveurs de routes intra-domaine* : serveurs de routes qui ne distribuent des routes externes qu'à l'intérieur d'un système autonome.
2. *les serveurs de routes inter-domaine* : serveurs de routes qui distribuent des routes externes entre plusieurs systèmes autonomes.

Il ne peut y avoir d'échange de routage entre deux serveurs de routes de types différents. Pour que le processus de décision puisse fonctionner correctement lors de l'apprentissage d'une route par le client d'un serveur de routes, il faut que celui-ci sache d'où provient cette annonce. Pour cela, il faut que deux conditions soient remplies :

1. le serveur de routes doit indiquer à son client qui est à l'origine de l'annonce. Ceci se fait par l'intermédiaire d'un attribut supplémentaire appelé ADVERTISER dans lequel est mis l'adresse du border router. Cet attribut est optionnel et non transitif et son code est 255.
2. le client doit pouvoir appréhender l'attribut ADVERTISER.

[22] Pour autant qu'il y ait plus d'une session iBGP dans l'AS.

Si le client reçoit un message UPDATE de la part d'un serveur de routes intra-domaine, alors la valeur du champ ADVERTISER doit être une adresse d'un border router du même AS que le client. Si le client reçoit un message UPDATE de la part d'un serveur de routes inter-domaine, alors le client peut déterminer de quel AS provient ce message en regardant le dernier numéro d'AS inséré dans le champ de l'AS-PATH de ce même message.

1.5.2 Réflecteur de routes

Le principe de la réflexion de route [Che00b] est de retirer la restriction selon laquelle un routeur qui apprend une route par un pair iBGP ne peut la distribuer à un autre pair iBGP. Contrairement aux serveurs de routes, un réflecteur de routes reste un pair BGP à part entière et effectue donc à ce titre les filtres à l'entrée et à la sortie mais aussi le processus de décision. Pour ce faire, il faut différencier deux catégories de routeurs :

- le réflecteur de routes : c'est via ce routeur que des annonces iBGP vont être redistribuées à d'autres pairs iBGP. Celui-ci doit connaître les routeurs iBGP avec lesquels il peut enfreindre la règle première qui interdit cette redistribution. Il est en effet important de signaler que cette infraction n'est pas faite avec tous les routeurs iBGP !
- réflecteur de routes client : ces routeurs fonctionnent comme des routeurs BGP normaux. C'est avec ces routeurs que le réflecteur de routes pourra distribuer des routes apprises via une session iBGP.

Il est important de signaler que si un réflecteur de routes a plusieurs réflecteurs de routes clients, ceux-ci ne doivent pas non plus être interconnectés entre eux. Car le mécanisme marche dans les deux sens. En effet, si un réflecteur de routes apprend une route de la part d'un pair iBGP normal, il redistribuera cette route à ses reflecteurs de routes clients. Et si il reçoit une route de la part d'un de ses réflecteurs de routes clients, il redistribuera aussi cette route aux autres réflecteurs de routes clients. L'attribut NEXT-HOP ne doit absolument pas être changé par le réflecteur de routes qui redistribue cette route sinon on n'obtiendrait pas les mêmes conditions de routage comme lors d'un maillage complet.

Deux attributs ont été ajoutés pour pouvoir faire fonctionner ce mécanisme ou plutôt pour éviter certaines erreurs de bouclage à l'intérieur de l'AS.

1. L'attribut ORIGINATOR-ID : Longueur 4 octets. Cet attribut a comme code 9 et est un attribut optionnel non transitif. Lorsqu'un réflecteur de routes apprend une route par l'un de ses réflecteurs de routes clients, il insère, dans le champ ORIGINATOR-ID, l'ID de ce reflecteur de routes client. Lorsqu'il annonce cette route aux autres pairs, il vérifie que l'ID du pair pour lequel il veut effectuer cette annonce n'est pas le même que celui inscrit dans le champ ORIGINATOR-ID. Autrement dit, il évite d'annoncer une route à un réflecteur de routes client si c'est celui-ci qui lui a appris.
2. L'attribut CLUSTER-LIST : La longueur de ce champ est un multiple de 4 octets. Il a comme code 10, il est optionnel et non transitif. Il est utilisé pour éviter à un réflecteur de routes de revoir passer une route apprise par l'un de ses réflecteurs de routes clients. Ce champ enregistre le chemin pris dans la hiérarchie de la réflexion de routes. Un cluster est composé d'un réflecteur de routes et de ses réflecteurs de routes clients. Lorsqu'un réflecteur de routes annonce une route à un pair non client, alors il insère dans cette liste de cluster son ID de cluster. Si le réflecteur de routes voit passer une annonce avec l'ID de son cluster, il rejette cette annonce.

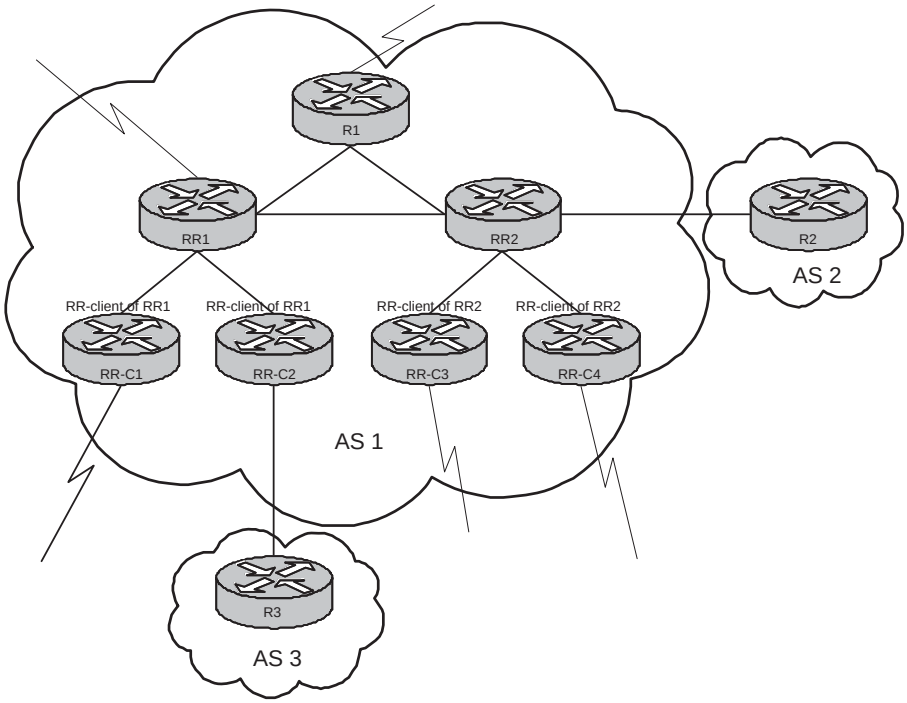


FIG. 1.18 – *Exemple d'architecture avec réflecteurs de routes*

Comme on peut le voir sur la figure 1.18, nous avons, dans l'AS1, un arbre dont la racine (R1) est un pair BGP tout à fait normal, ses deux fils (RR1, RR2) sont des réflecteurs de routes et chacun de ceux-ci ont à leur tour deux autres fils qui sont des RR-clients (RR-C1, RR-C2 pour RR1 et RR-C3, RR-C4 pour RR2). Le nombre de sessions entre les pairs iBGP est, comme on peut s'y attendre, bien moindre - 7 sessions - que si l'on avait dû effectuer un maillage complet entre ceux-ci - qui aurait compté 21 sessions.

Si une route est annoncée via la session eBGP de R2 vers RR2, celui-ci va se charger de l'injecter dans le maillage complet iBGP, donc aux routeurs R1 et RR1, et à ses clients RR-C3 et RR-C4[23]. Jusqu'ici, tout se passe comme avec des sessions normales iBGP. Mais le routeur RR1, ayant reçu une annonce de RR2, va lui aussi se charger d'injecter cette route à ses RR-clients. Une fois que ces deux actions sont faites, alors tous les routeurs ont pris connaissance de la route annoncée par le routeur R2, comme si cela s'était passé dans un maillage complet iBGP.

De même, dès que le routeur R3 annonce une route à RR-C2, celui-ci fait suivre cette route à RR1. Comme RR1 est un réflecteur de routes, celui-ci doit faire suivre cette route dans le maillage complet iBGP. Mais, pour que tous les routeurs aient appris cette route, celui-ci doit aussi l'annoncer à RR-C1 en faisant attention de ne pas l'annoncer à nouveau à RR-C2, ce qui peut se vérifier au moyen de la valeur de l'attribut ORIGINATOR-ID.

[23] On sous-entend que celle-ci doit être annoncée. Donc celle-ci est meilleure que toute autre route vers la même destination et elle n'a pas été filtrée en entrée ou en sortie. Cette remarque reste valable, en général, pour tout le reste du travail.

1.5.3 Confédération d'AS

Le principe d'une confédération de systèmes autonomes [Scu01] est de diviser un AS en plusieurs petits AS. Ceux-ci restent invisibles aux AS externes à cette confédération. Pour pouvoir faire cela, deux types de segments supplémentaires de l'AS-PATH ont été définis, l'AS_CONFED_SET et l'AS_CONFED_SEQUENCE qui permettent de distribuer les routes entre les systèmes autonomes d'une même confédération. Une fois que la route est annoncée à un pair extérieur à cette confédération, l'AS-PATH ne contient plus les segments rajoutés par cette confédération. De plus, comme il s'agit de sessions iBGP, les routeurs annonçant une route à un autre AS faisant partie de la confédération ne changent pas les valeurs du NEXT-HOP, du LOCAL-PREF et du MED.

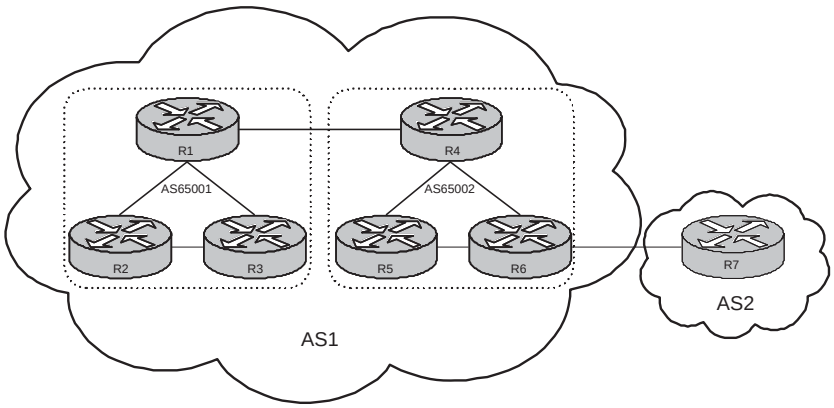


FIG. 1.19 – *Exemple d'architecture avec confédérations*

En prenant pour exemple la figure 1.19, on peut voir que deux AS sont définis dans l'AS1. Les numéros de ces AS sont des numéros réservés pour un usage privé interne à chaque AS. Lorsque le routeur R7 annonce une route à R6, celui-ci fait suivre cette annonce dans le maillage complet de l'AS65002. Une fois que le routeur R4 apprend cette route, il l'annonce aussi à R1 comme s'il s'agissait d'une session eBGP. Cependant, le message UPDATE diffère un peu car l'AS-PATH contient un segment du type AS_CONFED_SEQUENCE ayant pour valeur le numéro de l'AS65002 et les valeurs du NEXT-HOP, du LOCAL-PREF et du MED - s'il est présent - restent inchangés par rapport à leur valeur en entrant dans l'AS1.

De même, si le routeur R1 annonce une route à R4, celui-ci fait suivre cette annonce au routeur R6. A ce moment-ci, R6 doit annoncer cette route à R7. Avant de faire cette annonce, R6 doit enlever tout segment dans l'AS-PATH du type AS_CONFED_SEQUENCE ou AS_CONFED_SET et rajoute un segment du type AS_SEQUENCE avec comme valeur le numéro du véritable AS, dans ce cas-ci 1.

Il est à noter qu'un mécanisme similaire pouvait déjà être utilisé en subdivisant un même AS en plusieurs autres AS, ce qui réduisait aussi le nombre de connexions à établir. Seulement, il y a plusieurs conséquences à ce scindement que la confédération de systèmes autonomes résout. Tous les nouveaux AS créés sont vus par les autres AS. Ce qui implique que l'AS-PATH peut devenir plus long. Ce qui demande une plus grande gestion de la part des routeurs et une plus grande attention à porter pour la configuration de ceux-ci. D'autre

part, le forwarding des paquets s'en trouve à nouveau affecté et celui-ci peut devenir encore un peu moins efficient.

Ces trois techniques rendent de grands services aux administrateurs de larges réseaux. Pourtant, il est possible de créer des conditions pour que surviennent des oscillations de routes[24], c'est-à-dire une route qui se voit annuler implicitement sans cesse entre deux routeurs. Cette oscillation de route n'est pas due à la seule configuration des réflecteurs de routes et de ses clients ou des confédérations mais aussi aux différentes valeurs que peut prendre le MED pour un préfixe donné et circulant dans l'AS.

1.5.4 Loopback address

La deuxième différence importante entre eBGP et iBGP se situe au niveau de la topologie physique et logique d'un réseau. Lorsque deux pairs eBGP sont connectés c'est en général via une ligne physique directe, ce qui implique que lorsque la ligne est coupée, il n'y a plus moyen de communiquer entre eux. Mais lorsqu'il s'agit de pairs iBGP ce n'est pas forcément le cas. En effet, si l'on fait attention à la figure 1.20, on peut remarquer

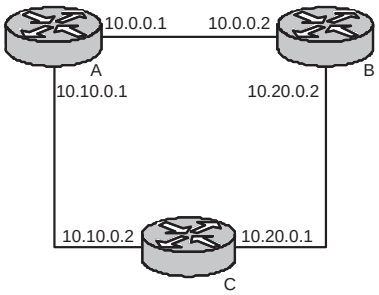


FIG. 1.20 – *Liens entre pairs iBGP*

que si l'interface 10.0.0.1 du pair A tombe en panne, il est quand même possible de joindre celui-ci par l'interface 10.10.0.1. Or si le pair B est configuré pour communiquer avec le pair A via l'interface 10.0.0.1, cette communication ne sait plus se faire. Ce qui implique que la distribution des routes ne se fait plus entre le pair A et le pair B et pourtant elle continue à se faire entre le pair A et le pair C ! Comme l'annonce de routes apprises par un pair iBGP ne peut être réannoncée à un autre pair iBGP, les tables de routage ne sont plus cohérentes entre elles. Donc, il faut trouver un moyen pour que cette communication reste possible tant qu'il existe un chemin pour joindre ce routeur. C'est via l'IGP que cela peut se faire et pour cela il suffit d'attribuer une adresse spéciale au routeur, adresse que l'on appelle *loopback address*. C'est une adresse unique qui est distribuée par l'IGP dans le réseau de l'AS. Puisque le routeur ne communique plus, en interne, qu'avec cette adresse, il est dès lors possible d'avoir un lien qui se coupe, et après réaction de l'IGP, s'il existe encore un chemin entre les deux routeurs, la communication entre les deux pairs peut continuer.

1.6 Instabilité des routes annoncées

Ce problème a déjà été mentionné au début de ce chapitre, BGP est un protocole très efficient si les routes annoncées restent stables. Cependant, pour diverses raisons comme

[24] Voir le draft suivant [Ret02] pour une explication textuelle détaillée, dont une démarche à suivre pour éviter ce problème et, pour une représentation plus visuelle, voir [ESN01c], [ESN01a], [ESN01b].

l'instabilité de liens ou des erreurs d'implémentation du protocole [Wet02], la stabilité des routes n'est pas de mise. Ceci implique que des routes se voient annoncer puis annuler régulièrement. Cette instabilité induit un gaspillage de la bande passante du réseau et du temps processeur des routeurs BGP. Plusieurs des techniques permettant de réduire l'instabilité des routes vont être présentées dans les paragraphes suivants.

BGP flap dampening Ce problème a été en partie résolu par l'introduction dans la spécification du protocole BGP-4 de deux timers fixes. Le premier timer *MinRouteAdvertisementInterval* détermine le temps minimum qui doit s'écouler entre deux annonces d'une même route. Le deuxième timer *MinASOriginationInterval* détermine le temps qui doit s'écouler entre deux messages UPDATE annonçant un changement dans le propre AS du pair BGP faisant l'annonce.

Ensuite, une extension est venue s'ajouter pour pallier aux défaillances de la méthode des timers fixes. Cette extension s'appelle le **route flap dampening** [Gov98]. Cependant, cette technique ne s'applique qu'aux pairs eBGP et non aux pairs iBGP pour éviter des bouclages dans le réseau. Le principe de cette extension est très simple, à chaque fois qu'une route est annulée, on incrémente un compteur. Quand l'état de la route ne change pas, on décrémente ce compteur de manière exponentielle et ce, avec la possiblité d'effectuer ce décompte de manière différente selon que la route est joignable ou pas. En fait, si la route n'est pas joignable, le décompte peut se faire de manière plus lente ou ne pas se faire du tout. Il suffit donc de définir deux seuils :

1. le seuil où l'on n'annonce plus le changement d'état de cette route. (*cutoff treshold*)
2. le seuil en-dessous duquel on recommence à annoncer une route. (*reuse treshold*)

En plus de ces deux seuils, il existe aussi un timer qui indique le temps maximum pendant lequel une route peut être supprimée (*maximum hold down time*). Passé ce délai, la route sera annoncée ou annulée quel que soit l'état de son instabilité.

Avec ces deux techniques, on peut masquer les changements d'états trop fréquents de routes dans l'Internet global, et de ce fait, assurer une stabilité forcée des routes. Ceci permet donc d'épargner de la bande passante et du temps processeur de certains routeurs BGP. De plus, les auteurs se sont particulièrement attachés à ce qu'il n'y ait pas trop de calculs à effectuer pour implémenter cette technique au détriment d'un besoin d'un peu de mémoire vive supplémentaire.

Le message INFORM BGP n'est pas un protocole très tolérant aux fautes. A chaque fois qu'il doit envoyer un message NOTIFICATION, le protocole indique que la session doit être coupée. Cette coupure provoque une annulation de toutes les routes connues des deux pairs. Si ces routes sont installées dans le LOC-RIB, alors BGP doit aussi retrouver de meilleures routes. Et, lorsqu'il a trouvé ces nouvelles routes, il doit les annoncer à tous ses pairs[25]. Cette coupure de session implique donc, non seulement, les deux routeurs entre lesquels la session a été coupée mais aussi les routeurs environnants. Chacun d'eux doit utiliser du temps processeur et utiliser de la bande passante pour annoncer les changements. Ce nouveau message [War02] va permettre à BGP d'informer un pair qu'une erreur s'est produite sans pour autant couper la session. C'est une amélioration de la tolérance aux fautes de BGP mais celui-ci va aussi sans doute permettre d'épargner du temps processeur et de la bande passante du réseau.

[25] En faisant abstraction du route flap dampening.

Route refresh Grâce à cette capacité, un routeur BGP peut demander à un de ses pairs de lui redistribuer ses routes [Che00a]. Cet ajout permet d'éviter l'approche appelée *soft-reconfiguration* qui consiste à sauvegarder une copie de toutes les routes du routeur pendant des modifications de politiques de routage. Une fois les changements effectués, on reconfronte les routes sauvegardées aux nouvelles politiques définies. Cette manière de procéder demande de la mémoire et du temps processeur.

Cette capacité doit être négociée lors de l'ouverture d'une session entre deux pairs.

Graceful Restart Mechanism Ce mécanisme de "*redémarrage grâcieux*" [Che02b] indique à un pair BGP que celui-ci peut retenir les routes pendant un redémarrage. Pour pouvoir supporter ce mécanisme, il faut le négocier lors de l'ouverture de session.

Le pair qui redémarre marque toutes ses routes comme *vieilles*. Il commence l'ouverture de la session avec l'autre pair. Une fois qu'il a reçu toutes les routes de celui-ci et que le processus de décision est terminé, il doit éliminer toutes les routes marquées comme *vieilles*. Lorsque ces trois opérations sont terminées, il peut alors recommencer lui-même à redistribuer ses routes aux autres pairs.

En ce qui concerne le routeur qui ne redémarre pas, deux scénarios peuvent se présenter :

1. soit il ne découvre pas la fin de session TCP avant de recevoir le message OPEN. Lorsqu'il reçoit le message OPEN, il peut déterminer qu'il s'agit d'un "redémarrage gracieux". Il doit alors tout simplement fermer l'autre session TCP[26] et marquer les routes de ce pair comme *vieilles*.
2. soit il découvre la fin de session TCP avant de recevoir le message OPEN et il ferme tout simplement la connexion TCP. Mais comme il sait que l'autre pair va redémarrer gracieusement, il n'efface pas les routes de ce pair et les marque comme *vieilles*.

Dans tous les cas, il doit, après avoir marqué les routes, redistribuer les routes de l'Adj-RIB-Out à ce pair.

Ce mécanisme permet de redémarrer un pair lorsque l'on en change sa politique de routage sans devoir couper complètement les sessions qu'il peut avoir avec d'autres pairs. Mais ce mécanisme peut aussi être utile lorsque le routeur redémarre à cause d'un crash de celui-ci.

1.7 Multiprotocol Extensions for BGP-4

Grâce à cette extension, BGP-4 va pouvoir distribuer des préfixes dont le format des adresses est autre que celui d'IPv4 [Rek02c]. De plus, il est prévu de pouvoir aussi distribuer des adresses multicast via ce mécanisme. Pour ce faire, cette extension doit fournir un moyen de distribuer des adresses d'un autre format dans le NLRI mais il faut aussi un autre moyen de distribuer le NEXT-HOP et l'AGGREGATOR qui tous deux doivent contenir des adresses IPv4. Il y a deux types d'attributs d'AS-PATH rajoutés à BGP-4 pour pouvoir distribuer ces informations :

- *MP_REACH_NLRI* : c'est l'attribut par lequel sont passés le NEXT-HOP, et le NLRI d'un autre type d'adresse que celui de l'IPv4.
- *MP_UNREACH_NLRI* : c'est par cet attribut que l'on va pouvoir annuler des annonces faites via le MP_REACH_NLRI.

[26] Ce qui n'est pas le comportement normal de BGP puisque celui-ci devrait normalement rejeter cette connexion.

1.8 Communautés et Communautés étendues

Le principe des communautés [Li96] est de marquer des routes qui partagent des propriétés communes. Ce marquage facilite la configuration des politiques de routage d'un AS en permettant d'effectuer certaines actions en fonction d'un marquage spécifique et non plus en fonction d'un numéro d'AS ou en fonction d'un espace d'adressage, ce qui rend leur configuration plus simple.

L'attribut est formé par 4 octets et il suit la syntaxe suivante :

AS Number	2 octets
0-65535	2 octets

Cet attribut est optionnel et transitif. Malheureusement, cette transitivité pollue aussi le routage de BGP [Bon02]. Cet attribut étant de plus en plus utilisé, et de par sa transitivité, une route peut avoir un grand nombre de communautés qui ne sont peut-être traitées qu'une fois - lors de leur premier passage dans un AS différent de celui qui attache une communauté ou si cette communauté n'a qu'une sémantique privée à un AS.

De plus, il n'y a pas moyen de passer beaucoup d'informations via une communauté et elles ne sont pas explicitement structurées. Ce qui fait que, pour chaque valeur d'une communauté, la sémantique de celle-ci peut varier.

Les communautés étendues [Rek02b] viennent améliorer les communautés sur trois points au moins :

1. L'attribut est élargi en taille. Et de ce fait, il y a moyen de mettre plus d'informations dans les communautés étendues.
2. Cet attribut est transitif mais on peut indiquer qu'il ne faut pas le distribuer à d'autres AS. Ce qui limite sa transitivité et, donc, peut de ce fait aider à la réduction de la pollution des communautés.
3. Les communautés étendues sont structurées ce qui les rend plus compréhensibles et leur configuration en devient encore plus simple que pour les communautés.

Type high	1 octet
Type low (optional)	1 octet
Value	6 or 7 octets

FIG. 1.21 – *Format des communautés étendues*

Plusieurs utilisations sont déjà définies pour les communautés étendues. On peut passer via les communautés étendues des numéros d'AS sur 2 octets ou sur 4 octets [27] plus une information quelconque dans les 4 ou 2 octets restants, une adresse IPv4 plus une information quelconque dans les 2 octets restants, indiquer l'origine d'une annonce, indiquer la bande passante d'une connexion entre deux routeurs directement connectés...

[27] Voir le draft [Che02a] décrivant les numéros d'AS sur 4 octets.

2 Le logiciel *Zebra*

Le logiciel utilisé s'appelle Zebra (version 0.92a)[28], il supporte les protocoles de routage dynamiques comme RIP ou OSPF que ce soit pour IPv4 ou IPv6 et bien sûr BGP-4.

Le logiciel *Zebra* est en fait un assemblage de plusieurs démons qui permettent de gérer indépendamment les différents protocoles de routage dynamiques. Ce qui revient à dire que lorsque l'on a uniquement besoin d'un routeur OSPF, on ne fait tourner que le démon OSPF et pas les autres en même temps. De plus, comme on peut le voir sur la figure 2.1, pour pouvoir bénéficier des interconnexions entre ces différentes implémentations de protocoles, *Zebra* sert d'interfaces entre les démons et la table de routage du kernel. Chacun de ces démons dispose de sa propre table de routage et indique à *Zebra* que ce qu'il doit savoir et vice et versa. De plus, chacun des démons, représentant un protocole, dispose également d'un système de lock différent pour sa table de routage[29].

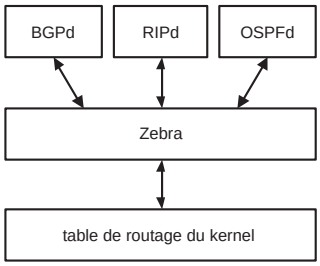


FIG. 2.1 – *Architecture globale de Zebra*

Pour le sujet de ce rapport nous n'allons utiliser que la partie BGP de ce logiciel, dénommée BGPd. Par la suite, il sera fait mention soit de *Zebra*, soit de BGPd pour parler du même logiciel. Dans cette partie, l'architecture sur laquelle repose BGPd et la manière dont les traitements importants sont effectués par celui-ci sont décrites. Cette description fait référence à certaines variables utilisées par BGPd. La description de ces variables contenues dans des structures spécifiques de BGPd se trouve à la fin de ce rapport en Annexe A.

2.1 L'architecture de BGPd

Dans l'architecture générale Figure 2.2, nous pouvons remarquer que ce démon fonctionne principalement avec deux types de threads. Le premier thread s'occupe de lire les données arrivant du réseau. Chaque paquet lu est mis dans la structure de l'instance BGP concernée. Ensuite, pour chaque paquet lu, le thread effectue les opérations adéquates, en fonction du type de paquet. Une fois ces opérations terminées, si un message doit être envoyé à un pair BGP, il sera mis dans une liste chaînée, celle-ci étant dépendante de la structure du pair concerné. Le deuxième thread s'occupe quant à lui d'envoyer tous les paquets de cette liste chaînée de manière FIFO au pair BGP concerné.

Le nombre de thread envoyant les messages est dépendant du nombre de pairs BGP avec lesquels Zebra a établi une connexion et est dans l'état *Established*.

[28] http ://www.zebra.org

[29] Mrtd, par exemple, ne fonctionne pas du tout ainsi. Il lui est impossible de ne faire tourner que OSPF, il ne dispose pas d'une table de routage pour chaque type de protocole et n'a pas non plus un système de lock indépendant

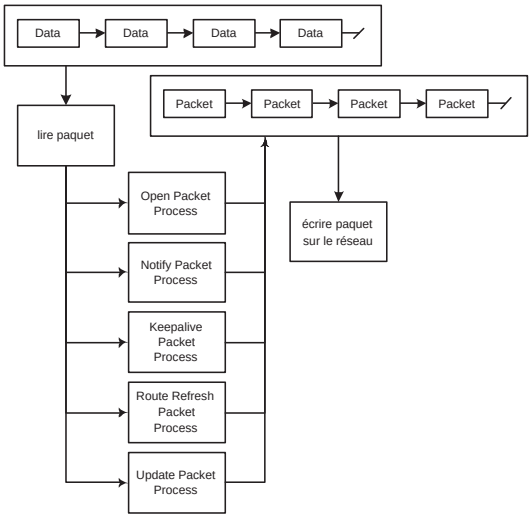


FIG. 2.2 – *Architecture générale de BGPd*

2.1.1 Ouverture de session

Lorsque deux pairs se connectent entre eux, ils s'envoient des messages OPEN et KEEPALIVE pour établir le début de la communication. Ceci leur permet de s'échanger certaines informations pour le bon déroulement des opérations par la suite. Une fois que *Zebra* est dans l'état *Established*, il va commencer à envoyer sa table de routage à l'autre pair. Pour ce faire, il lui suffit de parcourir sa LOC-RIB représentée par une structure route_table[30] dans laquelle sont toutes les routes actives. Pour chaque route, il doit déterminer quels sont les attributs à envoyer au pair. Ceci se fait par le parcours de la structure bgp_info[30] qui lui permet de déterminer quels sont les attributs qui ont été sélectionnés comme meilleurs. Enfin, *Zebra* envoie cette route à l'autre pair BGP avec qui il vient d'initier une discussion. *Zebra* répète cette opération tant qu'il y a encore une route dans sa LOC-RIB.

2.1.2 Traitement d'un message UPDATE

Les figures 2.3 et 2.4 montre un plan plus détaillé dont le message UPDATE est traité par *Zebra*.

Traitement d'un message UPDATE annulant des routes La figure 2.3 montre comment les WITHDRAW sont traités dans *Zebra*. Lorsqu'un message de type UPDATE est reçu par *Zebra*, il analyse d'abord les champs remplis de ce message. La même fonction est utilisée pour l'analyse d'un champ contenant une liste de préfixes. Si cette fonction obtient une liste de préfixes pour laquelle n'est associée aucun attribut, cela lui permet de déterminer que c'est une liste de préfixes à annuler dans la table de routage. Pour annuler un préfixe dans une table de routage, il lui suffit d'effacer la structure bgp_info associée à ce préfixe et au pair qui lui a demandé d'annuler cette route. En fait, *Zebra* n'efface pas vraiment cette structure mais la marque comme invalide pour le moment et garde des statistiques pour les routes instables.

[30] voir Annexe A page 79

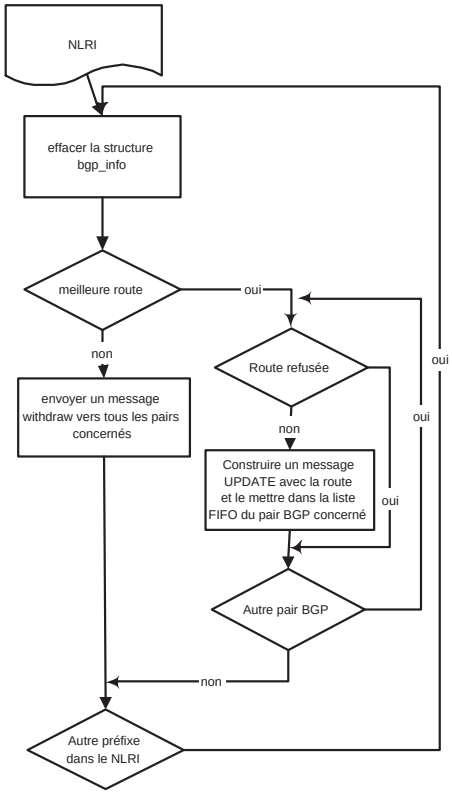


FIG. 2.3 – *Architecture du traitement des withdraw de BGPd*

Traitement d'un message annonçant des routes La figure 2.4 montre comment les annonces sont traitées dans *Zebra*. Si il y a des préfixes annoncés dans le message, alors on traite chacun de ceux-ci. Il y a 4 phases pour chaque préfixe se trouvant dans le message :

La première phase s'occupe de savoir si cette route doit être filtrée en entrée si cette route ne doit pas l'être alors on passe à la phase suivante sinon le traitement pour cette route s'arrête là.

La deuxième phase se trouve être le decision process. Si la route annoncée est meilleure que celle qui se trouvait déjà dans le LOC-RIB ou si elle est nouvelle cela veut donc dire qu'il faut l'annoncer aux autres pairs BGP avec lesquels on communique.

Donc pour chaque pair BGP, on regarde premièrement si la route ne doit pas être filtrée en sortie. Si celle-ci n'est pas filtrée alors, on construit un message UPDATE avec cette route et on le met dans la file d'attente (FIFO) du pair BGP concerné sinon le traitement s'arrête pour cette route.

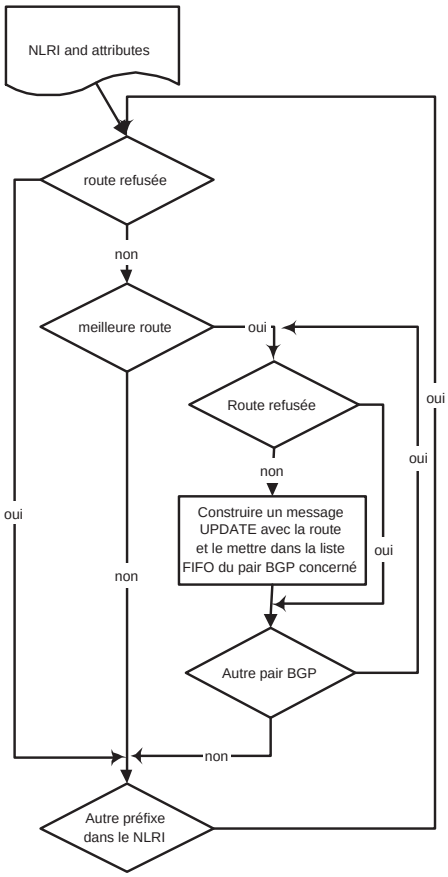


FIG. 2.4 – *Architecture du traitement des updates de BGPd*

2.1.3 Le Decision Process de *BGPd*

Le Decision Process décide quelle est la meilleure des routes connues de BGPd. Celui-ci est composé, dans l'ordre, des conditions suivantes :

1. *Weight check* : la meilleure route est celle qui a le plus grand poids. Il s'agit d'une métrique introduite par Cisco. Celle-ci n'est pas distribuée et doit donc être configurée manuellement pour chaque routeur.
2. *Local pref check* : Il faut d'abord vérifier que les deux routes ont une valeur pour la préférence locale. Si ce n'est pas le cas alors on leur attribue une valeur par défaut ou calculée. La meilleure route est celle qui a la plus grande valeur de préférence locale.
3. *Local route check* : La meilleure route est d'abord celle dont le type est ZEBRA_ROUTE_CONNECT puis ZEBRA_ROUTE_STATIC. Pour être plus précis, il s'agit, respectivement, des liens auxquels *Zebra* est directement connecté et des routes configurées statiquement. Si le type est le même alors la meilleure route est celle dont le sous-type est BGP_ROUTE_STATIC puis BGP_ROUTE_AGGREGATE, qui représentent les routes statiques de BGP ou des routes agrégées.
4. *AS path length check* : la meilleure route est celle dont la longueur de l'AS PATH est le plus court. (Si l'option de configuration ASPATH_IGNORE est faux)
5. *Origin check* : La meilleure route est celle dont l'origine est plus petite. Ceci s'explique par les valeurs attribuées aux constantes permises pour ce champ. IGP = 0, EGP =

1, INCOMPLETE = 2.

6. *MED check* : la meilleure route est celle dont la valeur du Multi Exit Disc est la plus petite.
7. *Peer type check* : la route est meilleure si le pair annonçant est du type eBGP et que l'autre est soit du type iBGP ou en confédération.
8. *IGP metric check* : la meilleure route est celle qui a la plus petite métrique IGP.
9. Si les deux chemins sont externes, la meilleure route est celle reçue en premier.
10. *Router-ID comparison* : la meilleure route est celle dont l'ID du pair est le plus petit.
11. *Cluster comparison* : la meilleure route est celle dont la longueur du cluster est la plus petite.
12. *Neighbor Address comparison* : la meilleure route est celle dont l'adresse du pair annonçant est la plus petite.
13. *par défaut* le decision process prend la nouvelle route comme meilleure route.

3 Evaluation des performances de *BGPd*

Ce chapitre parle d'abord de l'intérêt de faire une évaluation de performances d'une implémentation du protocole BGP. La deuxième partie décrit succinctement l'outil créé permettant de faire les séries de tests. Ensuite, les tests sont présentés comprenant le testbed, le déroulement des opérations et les résultats interprétés associés à chaque test.

3.1 L'intérêt d'une évaluation de performance d'une implémentation du protocole BGP

Les performances d'un protocole tel que BGP influent directement sur l'utilisation de l'Internet. C'est notamment à partir de ce protocole que les datagrammes passant par l'Internet peuvent être correctement forwardés. Si le protocole était parfait, et, que ses implémentations l'étaient aussi, il n'y aurait quasiment aucun intérêt à effectuer ces tests. Malheureusement, on peut se rendre compte que de nombreuses discussions continuent à ce propos dans le monde des réseaux. Depuis la première version du protocole BGP-4, de nombreuses extensions ont fait leur apparition. Ces extensions ne représentent pas que des ajouts de fonctionnalités au protocole BGP, mais parfois aussi des changements de comportement pour éviter de générer trop de trafic ou pour éviter certains problèmes d'interconnexion. Ces ajouts peuvent eux-mêmes amener d'autres catégories de problèmes inconnus jusqu'alors[31]. En dehors de ces changements du protocole, un deuxième problème existe, lié à l'implémentation du protocole. Ceux-ci peuvent aussi générer/faire générer énormément de trafic aux pairs BGP. Le dernier problème dégradant les performances de BGP et/ou empêchant son bon fonctionnement réside dans la configuration même des pairs BGP[32]. Les problèmes cités influencent directement sur l'utilisation du réseau, comme le démarrage d'un pair qui, pendant cette période, génère beaucoup de trafic. Ceux-ci utilisent aussi des ressources comme le CPU ou la mémoire vive. Ces problèmes font l'objet de discussions pour permettre au protocole BGP de survivre encore quelques années. D'autres problèmes existent aussi de par l'ampleur que prend l'Internet ; le nombre croissant de routes fait que ce protocole demande de plus en plus de ressources CPU et de mémoire vive. Il n'est pas question ici d'améliorer un des problèmes précités mais plutôt d'essayer d'améliorer les performances d'une implémentation de ce protocole pour supporter relativement mieux ceux-ci.

Les tests effectués dans ce travail sont axés sur le temps de traitement des routes lors de la synchronisation des tables de routage quand deux pairs initient une discussion et lors de la réception des UPDATE. Ceci va permettre de mettre en évidence un point faible de l'implémentation de *BGPd* qui pourrait être améliorée par la suite.

3.2 Instrumentation de *BGPd*

Disposant du code de *BGPd*, on n'est pas obligé d'effectuer des évaluations de performance comme si on disposait d'une boîte noire. On peut se permettre de changer le code pour effectuer la collecte d'informations qui sont nécessaires. Malheureusement, comme on veut effectuer une évaluation des performances de *BGPd*, on doit aussi être attentif aux changements que l'on effectue dans le code pour rester le plus proche possible de la réalité du temps des traitements. Il est impossible d'enregistrer directement sur disque les informations collectées car cela prendrait beaucoup trop de temps. Il n'est pas non plus possible

[31] Voir par exemple [ESN01c], pour un problème de bouclage avec les réflecteurs de routes.
[32] Le papier suivant parle entre autres de ces 3 problèmes : [Wet02].

de les afficher à l'écran pour les mêmes raisons. Il faut donc créer une structure de données permettant de garder l'information à propos des time-stamps, de l'endroit où l'information a été collectée et en plus du paquet qui a retenu l'attention.

Cette section présente les informations qu'il est nécessaire de conserver durant les tests et comment celles-ci vont être sauvegardées.

Pour sauvegarder convenablement les informations nécessaires, on doit prendre en compte non seulement les informations utiles c'est-à-dire les informations sur lesquelles on peut travailler à la suite d'un test mais aussi le temps d'exécution du code inséré dans *Zebra* et la manière dont on va pouvoir indiquer à l'outil que le test débute et finit.

Avant de commencer la présentation de la structure permettant de conserver l'information tout au long d'un test, soulignons qu'il existe deux phases essentielles pour la sauvegarde de l'information :

1. la réception des messages UPDATE : pendant cette phase, les routes annoncées se trouvent dans une structure *stream*[33] appartenant à la structure du pair BGP qui envoie l'UPDATE.
2. l'envoi des messages UPDATE : pendant cette phase, les routes annoncées se trouvent dans une structure *stream_fifo*[34] appartenant à la structure du pair BGP qui doit recevoir les UPDATE.

informations utiles A première vue, certaines informations apparaissent évidentes à sauvegarder :

1. Lors de la réception des updates :
   - la route annoncée par le pair BGP
   - le moment auquel la route a été annoncée
2. Lors de l'envoi des updates :
   - l'adresse du pair BGP à qui l'on envoie l'information
   - la route que l'on envoie
   - le moment auquel on envoie l'update

On peut déjà noter que l'adresse du pair BGP ne doit être sauvegardée qu'une fois pour toutes les routes annoncées. Par contre, pour chaque route annoncée, il faut en sauvegarder le moment d'arrivée ou de départ. Ce qui implique qu'il faille un tableau ou une liste chaînée pour pouvoir sauvegarder ces informations. Cette structure est déjà satisfaisante dans le sens où toute l'information concernant les routes est prise en compte. Par contre, il manque encore certaines informations comme le temps d'exécution du code inséré ou la connaissance du moment où le test commence/finit.

temps d'exécution du code inséré Comme on est obligé d'ajouter du code à *BGPd* pour pouvoir sauvegarder l'information, même si ce temps est proche de la constante, pour un même test, il est préférable d'ajouter des time-stamps pour chaque portion de code rajoutée dans *BGPd*. Il va falloir rajouter un premier time-stamp au début de l'exécution de chaque portion de code rajouté et un deuxième lors de la fin de cette exécution. Cet ajout va permettre, lors du traitement des données, d'avoir une estimation plus fine du temps mis pour le traitement d'une route.

[33] voir en Annexe A page 80 pour une description de la structure *stream*
[34] voir en Annexe A page 80 pour une description de la structure *stream_fifo*

début et fin de test Pour certains tests, pendant la première phase de l'évaluation de performance, il faut annoncer des routes qui vont constituer le LOC-RIB initial de *BGPd*. Il est dès lors inutile de sauvegarder cette information dans la structure. Il faut donc un événement qui permette de savoir à partir de quel moment on va pouvoir commencer à conserver les informations dont on a besoin. Regardons d'un peu plus près le trigger déclenché par cet événement, on reviendra un peu plus tard sur le caractère de l'événement. Un trigger général ne suffit pas pour deux raisons :

1. Dans la première phase, il y a deux sous-phases importantes, une qui concerne le pair BGP recevant l'information, l'autre concernant le(s) pair(s) vers le(s)quel(s) cette information peut se propager. Donc, on peut commencer à sauvegarder les informations de réception quand on reçoit l'événement qui permet de déterminer que le LOC-RIB de *BGPd* est initialisé mais on ne peut absolument pas commencer à sauvegarder ces informations à partir de ce moment-là pour l'envoi des updates. En effet, il est certain que l'événement déterminant sera détecté avant que toutes les routes à annoncer ne soient annoncées aux autres pairs BGP (ne fut-ce que la route annoncée par cet événement).
2. Un seul trigger général pour la deuxième phase ne suffit pas non plus. L'envoi des updates se passe de manière indépendante pour chaque pair BGP puisqu'il y a une file de messages pour chacun de ceux-ci. Il faut donc que l'événement survienne pour chacun de ceux-ci. Ce qui implique aussi qu'il faille absolument au moins un trigger dans chaque structure des pairs BGP.

La solution reprise est celle comportant un trigger global pour la première phase de réception et un autre dans la structure.

Revenons maintenant sur le caractère de l'événement. Cet événement doit se retrouver dans les 2 phases décrites ci-dessus. Sachant que cet événement va être transporté avec un message UPDATE, c'est qu'il s'agit d'un attribut d'un préfixe annoncé. Il faut donc impérativement que cet attribut soit **transitif**. Il faut en effet, que cet attribut soit identifié en début de traitement lorsque l'on analyse les routes reçues et en fin de traitement lorsque l'on envoie les routes aux autres pairs BGP. L'attribut retenu est l'attribut *community*. Ceci surtout pour une facilité de configuration et de ses possibilités de valeurs. Il est assez simple de choisir une communauté non utilisée et de lui attribuer la sémantique de début de test. En pratique, il suffira de modifier une ligne de code définissant une constante représentant la valeur de cette communauté.

Il ne reste plus qu'à décrire la manière dont les informations vont être sauvegardées sur disque. Comme les données vont être enregistrées séparément pour chaque pair, il nous faut un descripteur de fichier pour chaque pair. Quant au moment de la sauvegarde, il survient lorsque le dernier message est envoyé. Pour chaque pair BGP communiquant avec *BGPd*, il existe un thread envoyant les messages à ces pairs. On peut donc créer un sémaphore initialisé à N (N étant le nombre de pairs vers lesquels on **envoie** les messages et pour lesquels on enregistre les informations). Il suffit ensuite de décrémenter de un le sémaphore à chaque fois que l'on *sait* que tout a été envoyé pour un pair spécifique. Une fois le sémaphore égal à zéro, cela signifie que le test est terminé et dès lors, on peut enregistrer l'information sur disque. Il reste à savoir comment on peut déterminer que *BGPd* a tout envoyé vers un pair BGP spécifique. Ceci est effectué par l'intermédiaire d'un événement du même type que celui déterminant le début du test, une route transportant comme attribut une communauté choisie.

On dispose dès à présent de tout pour sauvegarder les informations nécessaires à l'évaluation de performance. La figure 3.1 représente la structure que l'on vient de construire.

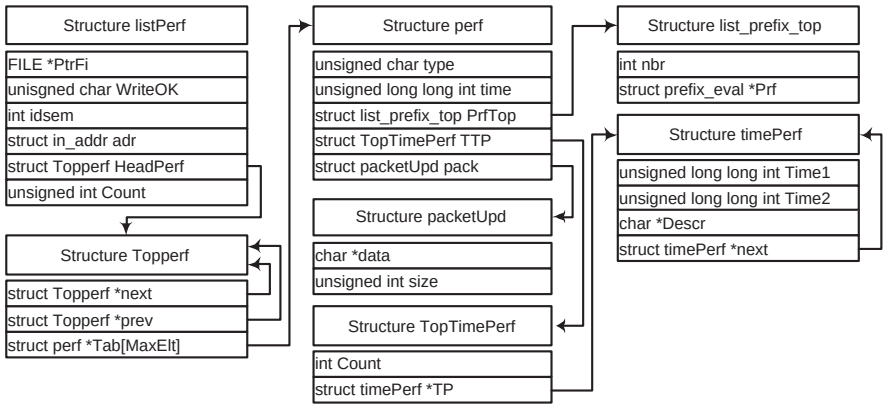


FIG. 3.1 – *Structure de sauvegarde des informations*

3.3 Résultats de l'évaluation de performance

3.3.1 Matériel utilisé

Pour pouvoir effectuer les tests, on a dû utiliser du matériel et certains logiciels :

- 3 PC Intel Celeron 300 MHz avec comme OS Linux noyau 2.4.18
  L'un d'eux fait tourner *BGPd*, les deux autres font tourner BGPsim [35].
- 2 routeurs Cisco 3600 (version d'IOS 12.2(7)) avec 64 Mo de mémoire vive et 6 interfaces Ethernet/IEEE 802.3.
- BGPsim[oMN01] : BGPsim est un utilitaire permettant de simuler un environnement complexe de routage BGP-4.

  BGPsim est utilisé de deux façons différentes :

  1. Son utilité première est d'effectuer une simulation de BGP-4. BGPsim permet de définir quelle discussion il tiendra avec son voisin. Il est dès lors possible d'envoyer des messages UPDATE vers *BGPd* pour pouvoir faire les tests de manière définie.
  2. Comme on peut contrôler la communication entre les deux hôtes, ceci va aussi nous permettre d'injecter une table de routage prédéfinie dans *BGPd*. Effectuer une évaluation des performances implique de la faire dans les conditions se rapprochant de la réalité. Pour cela, il faut que *BGPd* contienne déjà des routes dans sa table de routage. Cependant, contenir n'importe quelle route n'est pas non plus ce qui peut se rapprocher le plus de la réalité. Il est possible grâce à un outil fourni par Olaf Maennel[Mae02] de créer une table de routage réaliste. C'est donc avec son outil que l'on va créer la liste des routes à injecter dans *BGPd*. Cependant, dans les séries de tests qui vont suivre, les routes ne vont pas être tout à fait réalistes. Les attributs des préfixes vont tous être les mêmes. Ceci permet une meilleure lecture des graphiques et les caractéristiques de l'implémentation en sont soulignés. Par après, les mêmes séries de tests vont être effectuées avec des routes tout à fait réalistes[36].
- RTG : RTG, outil créé par Olaf Maennel, permet de créer des tables de routage et des updates réalistes. Afin de pouvoir créer ces données réalistes, il faut des fichiers de

[35] Disponible à l'adresse http ://www.merit.edu/ mrt.
[36] Les tests avec des routes réalistes sont présentés dans la section 4.5 page 64.

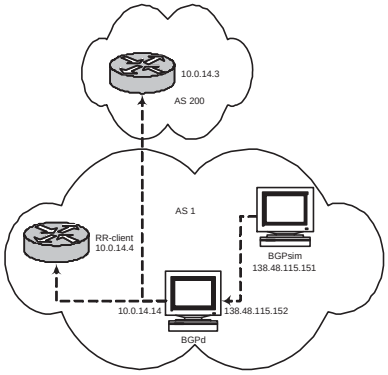


FIG. 3.2 – *Testbed de l'évaluation de la synchronisation de la table de routage*

configuration qui sont créés par un autre outil permettant de caractériser une structure dans le trafic BGP à partir de traces réelles. Cet outil permet de caractériser notamment la structure des préfixes et des attributs mais aussi l'instabilité détectées dans les traces analysées. Différents types d'instabilités peuvent être identifiés, l'instabilité due à un préfixe ou celle causée par une relation entre deux AS. Ces deux instabilités peuvent être identifiées selon le nombre d'UPDATE ré-annonçant des routes dans un intervalle de temps défini. Une fois les fichiers de configuration créés avec cet outil, on peut utiliser RTG. La première partie de RTG construit les préfixes de la table de routage - dont la taille est un paramètre à passer en argument à RTG -, la deuxième partie s'occupe d'associer chacun de ces préfixes à des attributs. RTG génére l'AS-PATH de manière à prendre en compte la topologie sous-jacente que cet attribut implique. Ceci se fait avec l'aide des paramètres du fichier de configuration. Enfin, les updates sont générés en fonction de la table de routage générée. Pour la génération de ces updates, certains événements sont pris en compte de manière aléatoire comme par exemple, des coupures de session d'un certain AS, l'instabilité d'un préfixe, le taux de nouvelle annonce d'un préfixe, la fréquence avec laquelle cela se fait...

3.3.2 Synchronisation de la table de routage

Lors du test de synchronisation de la table de routage, on va voir comment réagit *BGPd* lorsqu'il commence à communiquer avec un pair. On va observer le nombre de paquets envoyés par *BGPd*, le temps qu'il met pour envoyer cette table de routage et aussi le temps mis pour construire les paquets nécessaires à l'envoi de cette table.

Testbed La figure 3.2 montre que pour ce test, il a fallu deux PC, et les deux routeurs Cisco. Le premier PC fait tourner BGPsim dont l'utilité ici est de charger la table de routage pour *BGPd*, le deuxième PC fait tourner *BGPd*.

Déroulement Au départ, on charge la table de routage via le PC qui fait tourner BGPsim. Une fois cette table de routage chargée, on va pouvoir faire débuter la conversation entre *BGPd* et les routeurs Cisco. Pour pouvoir tester le temps de synchronisation, il faut que BGPd soit configuré de telle manière que lorsqu'il recevra sa table de routage, il ne la communique pas immédiatement au routeur Cisco, auquel cas nous prendrions des mesures

sur le flooding de messages UPDATE reçus. Donc, lors du lancement de *BGPd*, les routeurs Cisco ne sont pas dans sa configuration. Une fois que BGPd aura toute sa LOC-RIB initialisée, on ajoutera à la configuration de *BGPd* les routeurs Cisco. La configuration des routeurs Cisco est des plus élémentaires, puisque ceux-ci ne sont présents que pour initier une communication avec *BGPd*. Cette configuration ne comporte donc que la déclaration du pair BGP représenté par le PC faisant tourner *BGPd*. Dès que la synchronisation commence, quand la connexion passe dans l'état *Established*, le test commence. Une fois que le dernier message UPDATE est envoyé vers un des deux routeurs Cisco, le test s'arrête.

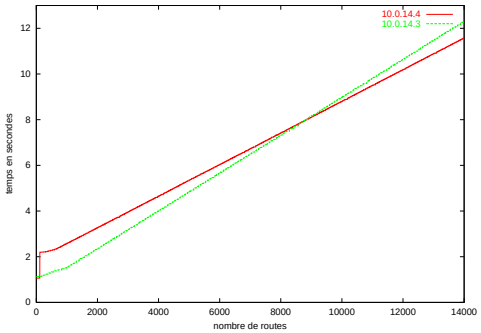


FIG. 3.3 – *Temps d'envoi de la table de routage*

Résultats *BGPd* synchronise sa table en parcourant l'arbre représentant la table de routage et, pour chaque préfixe, il recherche la structure bgp_info[37] qui lui permet de déterminer quelle a été l'annonce qui a été sélectionnée comme meilleure route. Une fois la route trouvée, *BGPd* construit un message UPDATE et le met dans la file d'envoi vers le pair avec lequel il débute la communication. Ce mécanisme fait que, pour chaque route à envoyer à ce pair, *BGPd* envoie un message UPDATE. La figure 3.3 représente le temps global de synchronisation de la table de routage. Les deux mesures prises pendant ce test sont :

1. T1 : le moment où *BGPd* trouve une route à envoyer,
2. T2 : le moment de l'envoi du message UPDATE.

Le temps total mis pour la synchronisation des 14000[38] routes à annoncer est de 11.6 sec pour le route-reflector-client et de 12.30 sec pour le pair eBGP. L'envoi des paquets est fait de façon linéaire avec comme équation

$$y = 6,92097\ 10^{-4} \times x + 1.88\ (pour\ x > 1000)$$

pour le routeur route-reflector-client et

$$y = 8,282715\ 10^{-4} \times x + 0,71\ (pour\ x > 1000)$$

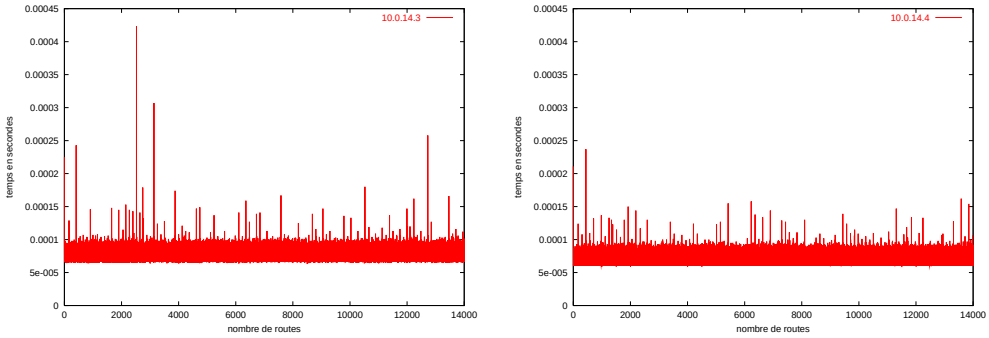
pour le routeur eBGP.

Cependant, on peut remarquer que cette croissance linéaire ne débute qu'aux alentours de la millième route et que le temps qui s'est écoulé entre le moment où BGPd a trouvé la

[37] Voir Annexe A page 79.

[38] Le nombre de routes actives dans les routeurs BGP principaux environne les 150000 routes. Dans notre labo, les routeurs Cisco n'ayant qu'une mémoire vive limitée à 64Mo, il nous est impossible de leur faire appréhender 150000 routes. Il a donc fallu se limiter à 14000 routes.

première route à envoyer et l'envoi de celle-ci est de 1,06 sec pour le route-reflector-client et 1,13 sec pour le pair eBGP. De plus, en ce qui concerne le RR-client, on peut distinguer qu'un arrêt des envois se produit après la route numéro 114. Cet arrêt est provoqué lorsque *BGPd* a fini de mettre les messages UPDATE pour le RR-client dans sa file d'envoi. En fait, *BGPd* commence à ce moment-là à parcourir à nouveau cette table pour le pair eBGP et ne fait que ça. Si l'on omet le temps d'arrêt d'envoi du RR-client, la première route aurait dû être envoyée après 0,71 sec, que ce soit pour le route-reflector-client ou pour le pair eBGP. La raison pour laquelle ces mille premiers paquets ne sont pas envoyés selon la même croissance n'a pas été identifiée.



(a) *Temps de construction des messages UPDATE pour le pair eBGP*

(b) *Temps de construction des messages UPDATE pour le route-reflector-client*

FIG. 3.4 – *Temps de construction des message UPDATE*

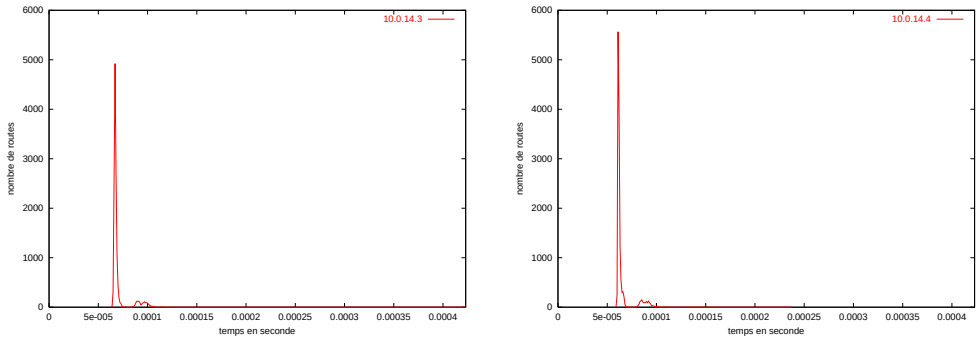
Les figures 3.4(a) et 3.4(b) représentent le temps mis par *BGPd* pour mettre tous les messages dans la file d'envoi pour, respectivement, le pair eBGP et le route-reflector-client. Les deux prises de temps sont :

1. T1 : le moment où *BGPd* trouve une route à envoyer,
2. T2 : le moment où le message UPDATE est mis dans la file d'attente.

Le temps total est de 0,91 sec pour une moyenne de 0,000065 sec pour le route-reflector-client. En ce qui concerne le pair eBGP, le temps total est de 0,99 sec pour une moyenne de 0,00007 sec. Le temps maximal du route-reflector-client et du pair eBGP sont, respectivement, de 0,000237 et de 0,000423, dû certainement à une prise de contrôle de l'OS. On peut remarquer sur les figures 3.5(b) et 3.5(a) que leur distribution est bien uniforme et cela se vérifie par leur écart-type qui est, respectivement, de $7,99\ 10^{-8}$ et $8,84\ 10^{-8}$.

On peut remarquer qu'il existe des temps de traitement en quantité non négligeable entre les 0,00008 et 0,0001 seconde. Cependant, la raison pour laquelle ces temps existent n'a pas été identifié.

Finalement, on constate que la plus grande partie du temps passé par *BGPd* pour la synchronisation de la table de routage réside dans l'envoi des messages à destination du pair BGP. Ceci est certainement dû à la multiplication des accès aux réseaux.



(a) *Distribution pour le pair eBGP*

(b) *Distribution pour le route-reflector-client*

FIG. 3.5 – *Distribution du temps de construction des messages UPDATE*

3.3.3 Traitement des UPDATE par *BGPd*

Cette série de tests se concentre sur le traitement des UPDATE par *BGPd*. Sont abordés dans ces tests, le nombre de messages UPDATE renvoyés par *BGPd*, le temps mis par celui-ci pour traiter totalement un message UPDATE, le temps mis pour le traiter sans tenir compte de l'envoi des UPDATE générés et le temps d'activité pour traiter chaque route.

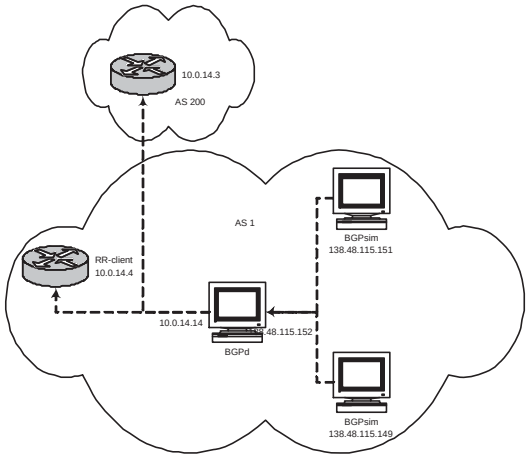


FIG. 3.6 – *Testbed de l'évaluation du traitement des UPDATE*

Testbed La figure 3.6 nous montre que le testbed est composé de trois PC, et deux routeurs Cisco. Le premier PC fait tourner BGPsim, et remplira la table de routage de *BGPd*, le deuxième enverra des messages UPDATE pour lesquels les time-stamps sont pris et le troisième fait tourner *BGPd*. Les deux autres routeurs sont configurés pour communiquer avec *BGPd*, l'un est configuré comme Route Reflector-client et l'autre est un pair eBGP. On peut constater que les deux PC faisant tourner BGPsim sont configurés pour être des pairs iBGP et non eBGP. De cette manière, BGPd n'essaie pas de redistribuer les routes de l'un vers l'autre pour éviter d'interférer avec l'évaluation du temps des traitements.

Déroulement La première étape de ce test consiste tout d'abord à remplir la table de routage de *BGPd*. Ensuite, une fois cette table de routage remplie et la synchronisation de cette table avec les deux routeurs Cisco finie, on commence à envoyer des messages UPDATE vers *BGPd*. Ces messages UPDATE contiennent tous de meilleures routes (AS-PATH plus court) que celles qui constituent la table de routage. Le départ du test commence lors de la réception du premier message UPDATE et finit selon le cas, lors de l'envoi du dernier message UPDATE ou du traitement de la dernière route à envoyer.

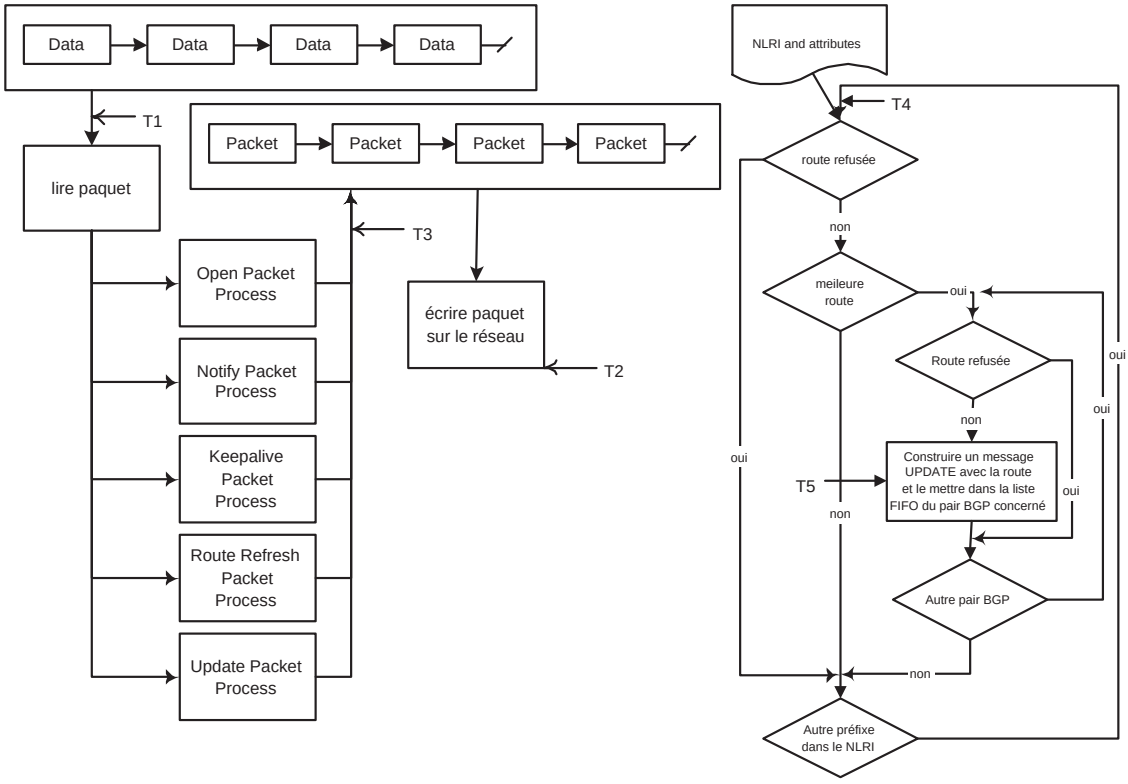


FIG. 3.7 – *Time-stamps du traitement des UPDATE*

Résultats De par le mécanisme observé sur la figure 3.7, *BGPd* traite les routes une par une, de l'extraction de cette route de l'UPDATE reçu jusqu'à ce qu'elle soit mise dans un UPDATE pour la faire suivre aux autres pairs BGP. Ceci permet de constater que, pour chaque message UPDATE contenant N routes annoncées, *BGPd* génère N messages UPDATE[39].

La figure 3.8 représente le temps de traitement global d'une suite de messages UPDATE reçus par *BGPd*. Les deux mesures prises pour ce test sont (figure 3.7) :

1. T1 : l'arrivée d'un paquet contenant N routes,
2. T2 : le départ d'un message UPDATE à destination d'un des deux routeurs Cisco contenant une route.

On obtient donc j time-stamps T2 correspondant au nombre de routes à réannoncer aux pairs BGP et i time-stamps T1 correspondant aux messages UPDATE reçus. L'équation

[39] Si ces routes ne sont pas filtrées et si celles-ci sont meilleures que celles déjà existantes dans la table de routage.

par laquelle nous trouvons le temps de traitement global d'un UPDATE est la suivante : $T2_j - T1_i$ (où i est le numéro du message dans lequel se trouve la route correspondante au time-stamp $T2_j$). On peut constater ici l'asynchronisme entre le traitement de la route et l'envoi de cette route vers les autres routeurs par les moments décroissants de ce graphique. Cela veut dire que lorsque le début du traitement d'un paquet commence, les derniers messages UPDATE à annoncer n'ont pas encore été envoyés. Le temps total pris pour envoyer toutes les routes aux deux pairs est de 12,9 secondes pour le RR-client et de 12,1 pour le pair eBGP. Le plus grand écart qui existe entre la réception d'un message UPDATE et l'annonce d'une des ses routes est de 10,49 secondes en ce qui concerne le RR-client. Le temps moyen mis par *BGPd* pour annoncer une route est de 6 secondes.

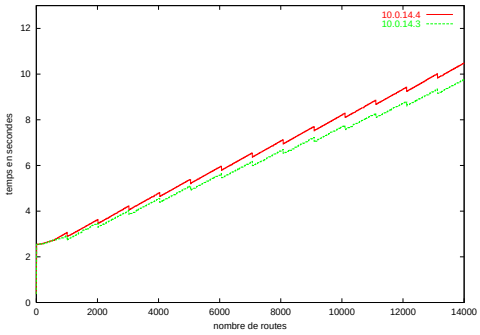


FIG. 3.8 – *Temps de traitement global*

La figure 3.9 représente le temps de traitement des routes cumulé par paquets. Comme *BGPd* n'envoie qu'un message par route à annoncer, il est intéressant de voir le comportement de *BGPd* sans tenir compte de l'envoi de ces messages. Ce test-ci prend deux autres time-stamps, comme indiqué sur la figure 3.7 :

1. T1 : l'arrivée d'un paquet contenant N routes,
2. T3 : le moment de la construction du message UPDATE pour une route à envoyer vers un autre routeur.

On obtient donc j time-stamps T3 correspondant au nombre de routes à réannoncer aux pairs BGP et i time-stamps T1 correspondant aux messages UPDATE reçus. L'équation par laquelle nous trouvons le temps de traitement est la suivante : $T3_j - T1_i$ (où i est le numéro du message dans lequel se trouve la route correspondante au time-stamp $T3_j$).

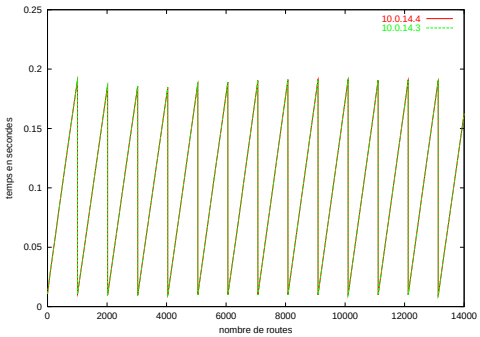
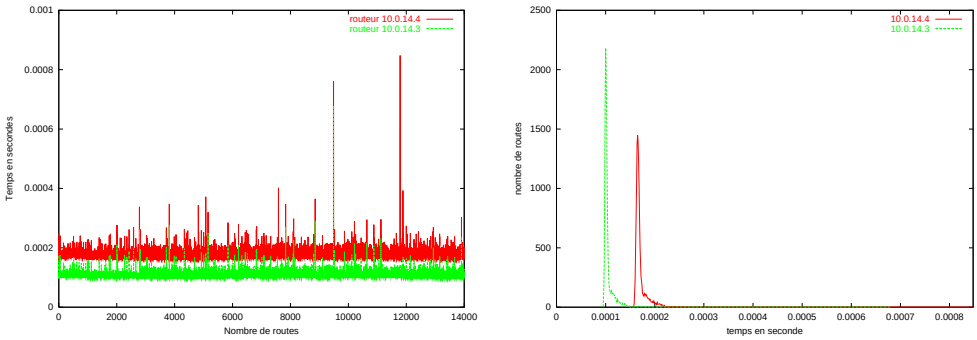


FIG. 3.9 – *Temps de traitement des routes*



(a) *temps de traitement jusqu'à la mise dans la file d'envoi*

(b) *Distribution du temps de traitement jusqu'à la mise dans la file d'envoi*

FIG. 3.10 – *Traitement des messages UPDATE jusqu'à la mise dans la file d'envoi*

La différence entre ces temps nous montre le temps de traitement d'une route jusqu'au moment où cette route, encapsulée dans un message UPDATE, va être mise dans la file d'attente FIFO de *BGPd* pour l'envoi. Ceci nous permet de constater que les temps sont beaucoup plus petits et donc, que la plus grande partie du temps passé par *BGPd* pour traiter les routes annoncées se situe lors de l'envoi des messages UPDATE vers les autres pairs avec lesquels il communique. Le temps de traitement moyen d'une route dans un paquet se situe aux alentours de 0,099 sec. Si nous additionnons le temps de traitement de chaque paquet nous arrivons à un temps de 2,515 secondes. Nous pourrions aussi croire que le traitement des routes s'effectue en même temps pour les deux pairs mais ceci n'est pas vrai, nous allons constater cette différence lors du test suivant.

La figure 3.10(a) représente le temps de traitement de chaque route jusqu'à la construction du message UPDATE. Avec ce test, nous pouvons constater l'écart du temps pris par la construction du message qui doit être envoyé vers les pairs BGP. Les deux temps pris sont, comme indiqué sur la figure 3.7 page 45 :

1. T4 : le début du traitement d'une route,
2. T5 : la fin de la construction du message UPDATE pour une route à envoyer vers un autre routeur.

Le temps moyen de traitement à destination du route-reflector-client est de 0,00017 sec et le pair eBGP, de 0,0001 sec. Leur maximum se situe à 0,000848 et 0,000679 pour respectivement, le route-reflector-client et pour le pair eBGP. Par contre, on peut voir sur la figure 3.10(b) que ces pointes ne sont que de très rares cas dus probablement à l'OS préemptif. L'écart-type qui est de $1,3\ 10^{-7}$ et $9,86\ 10^{-8}$ pour le route-reflector-client et pour le pair eBGP, indique que la variation des temps de traitement est très faible.

3.4 Conclusion

Les deux tests menés ont permis de mettre en évidence que lorsque *BGPd* doit envoyer des messages UPDATE, il n'essaye pas du tout de mettre plusieurs routes dans ces messages. Lorsque l'on regarde d'un peu plus près les temps de traitement, on peut remarquer que le temps passé par *BGPd* pour envoyer ces messages en constitue la majeure partie. Une des préoccupations actuelle est notamment le trafic généré par le protocole BGP-4 pour sa

viabilité à moyen terme. Or, avec *BGPd*, à chaque fois qu'il y a un démarrage avec un pair, toute la table est redistribuée route par route. Ce qui fait que pour une table de routage de 150.000 routes, cela prend deux minutes, sans perturbation, pour synchroniser la table de routage avec l'autre pair. On vient d'évoquer le cas d'un routeur qui démarre/redémarre à côté de *BGPd* mais c'est le même cas si un routeur démarre/redémarre, à deux sauts de lui. En effet, toute la table va lui parvenir dans des messages contenant N routes, si ce n'est pas une implémentation de BGPd, qui se trouve entre le routeur qui démarre/redémarre et lui, et *BGPd* les annoncera une par une. Non seulement, *BGPd* consomme du trafic inutilement, mais plus important encore pour son rôle, il consomme énormément de temps CPU pour envoyer ses messages. Ceci sans compter le temps que met l'autre routeur pour appréhender les messages qu'il reçoit de la part de *BGPd*. Une première bonne amélioration pour *BGPd* consiste donc en un changement de son comportement dans la manière dont il construit ses messages UPDATE.

4 Amélioration de BGPd

Comme on a pu le constater dans le chapitre précédent, une fois que *Zebra* reçoit un message UPDATE, contenant plusieurs préfixes annoncés, celui-ci renvoie un message UPDATE pour chaque préfixe contenu dans le message reçu. Lors du démarrage d'un pair devant communiquer avec *Zebra*, ces deux-ci vont devoir échanger leurs préfixes. Le trafic généré pendant cette phase est le plus lourd puisque dans la phase suivante, ils n'ont plus qu'à communiquer les changements par rapport à cette base de données de départ. Il peut être intéressant de changer cette caractéristique de *Zebra* pour que tout au moins il essaye d'envoyer plusieurs préfixes dans un même message UPDATE.

On se retrouve face à des choix d'implémentation pour effectuer ce changement de comportement de *BGPd* et plusieurs peuvent être liés entre eux :

- **le moment de la construction du message UPDATE** : deux possibilités sont envisageables. On peut soit construire les messages à la fin du processus de décision, soit le construire incrémentalement c'est-à-dire qu'à chaque fois que l'on connaît une route à annoncer à un pair, on l'insère dans un message UPDATE.
- **le moment de l'envoi des messages UPDATE aux pairs BGP** : on peut envoyer un message UPDATE dès que l'on sait qu'il est rempli ou attendre pour envoyer tous les messages en même temps. Le critère de remplissage peut être défini de manière différente. Par exemple, pour l'implémentation originale de BGPd, un message est rempli si celui-ci contient exactement une route. On peut aussi poser une contrainte sur les routes annoncées, en imposant que toute route annoncée par un message N doit obligatoirement être réannoncée avant toute route annoncée par un message N+i où i $>$ 0.

Plusieurs solutions sont envisageables pour changer le comportement de *BGPd*. Cependant, seulement deux de celles-ci ont été implémentées. Toutes les deux construisent leur message de manière incrémentale et toutes les deux envoient leur message sur base de la taille du message ; c'est-à-dire que lorsqu'un message est à sa taille maximale, alors ce message est envoyé directement vers le pair pour lequel on construit le message. La différence réside dans la manière de construire un message UPDATE comportant des routes à annoncer. Le protocole précise bien que l'on ne peut effectivement pas envoyer dans un même message UPDATE deux routes n'ayant pas les mêmes attributs. C'est donc sur ce point que les deux solutions diffèrent. La première solution envoie directement le message en construction si une nouvelle route à insérer n'a pas les mêmes attributs que ceux du message en construction. Tandis que la deuxième solution peut construire plusieurs messages UPDATE en même temps et choisir dans lequel la nouvelle route doit être insérée.

Dans la suite de ce chapitre, les deux solutions vont être présentées plus en détail. La série de tests de performance a été effectuée pour chacune des solutions et est présentée avec des comparaisons entre celles-ci et l'implémentation actuelle de *Zebra 0.92a*.

4.1 Tronc commun aux deux solutions

Avant de commencer à expliciter les solutions d'amélioration apportée, il est important de noter que le principe des deux solutions se base sur les mêmes contraintes qui vont être exposées ci-après. En effet, le changement du comportement de BGPd se limite à éviter d'envoyer trop de messages. Autrement dit, aucun changement ne porte sur les deux points suivants :

1. il est prévu par le protocole de pouvoir mettre à la fois dans le même message UPDATE des routes à annuler et des routes à annoncer,
2. il est possible de retarder l'envoi d'un message UPDATE pour essayer d'y insérer plus de routes parce que la taille de ce message est jugé trop petite. Ceci implique un mécanisme permettant d'éviter que BGPd n'attende trop longtemps avant d'envoyer ce message en attente.

Implémentation actuelle BGPd dispose de deux fonctions qui construisent chacune un message UPDATE. Le première fonction permet de créer un message annonçant une route et la deuxième crée un message annulant une route [40]. Les deux fonctions sont :

bgp_update_send (struct peer *peer, struct *p, struct attr *attr) Cette fonction crée un message UPDATE annonçant des routes. Elle y insère les attributs et le préfixe puis le met dans la file FIFO d'envoi du pair concerné.

bgp_withdraw_send (struct peer *peer, struct prefix *p) Cette fonction crée un message UPDATE annulant des routes. Elle y insère le préfixe à annuler puis le met dans la file FIFO du pair concerné.

Modification de l'implémentation Pour permettre à BGPd d'envoyer un message avec plus d'un préfixe dans le message UPDATE, il faut modifier ces deux fonctions. Si on veut pouvoir insérer plusieurs préfixes sans pour autant imposer la connaissance de tous les préfixes avant d'appeler une de ces deux fonctions, on doit logiquement créer trois fonctions plus élémentaires :

struct stream *bgp_packet_update_init () Cette fonction s'occupe uniquement de créer un message UPDATE de taille maximale, d'y insérer le header, le type du message et de mettre à zéro la longueur du champ WITHDRAW.

void bgp_packet_put_attribute (struct stream *s, struct peer *peer, struct attr *attr) Cette fonction n'est utile que lorsque l'on crée un message comportant des routes à annoncer et non à annuler. Ce qui confère à cette fonction l'utilité d'insérer les attributs dans le message UPDATE.

void bgp_packet_put_update_fifo (struct peer *peer, struct stream *s) Cette fonction est appelée lorsque le message doit être envoyé vers le pair BGP concerné. Celle-ci s'occupe de fixer la longueur du paquet, de l'ajouter dans la file FIFO du pair et d'indiquer à BGPd qu'il y a un message à envoyer.

Il ne reste plus qu'à redéfinir les deux fonctions permettant de construire un message, pour la première, annulant les routes et, pour la seconde, annonçant les routes. En ce qui concerne la fonction permettant d'annuler les routes, celle-ci a le même comportement dans

[40] BGPd ne construit jamais de messages UPDATE permettant à la fois d'annoncer une route et d'annuler une route. Ce comportement n'a pas été modifié par les implémentations effectuées.

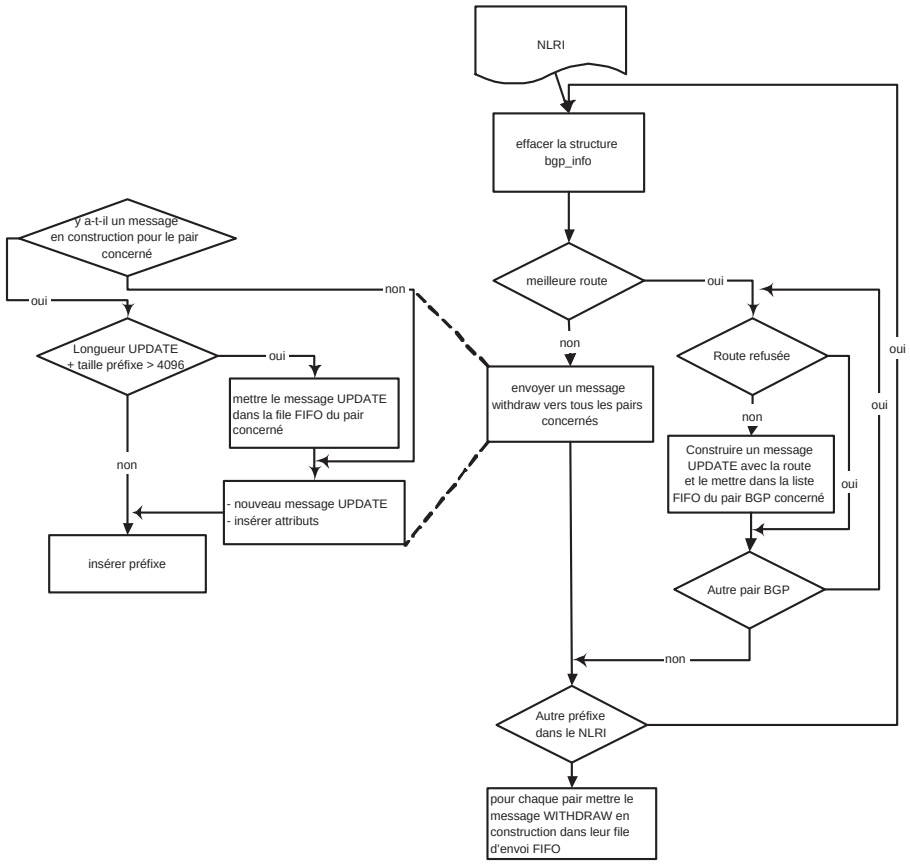


FIG. 4.1 – *Architecture de la construction des withdraw*

les deux implémentations effectuées. Ce qui n'est pas le cas pour la fonction annonçant les routes.

void bgp_withdraw_build (struct peer *peer, struct prefix *p) Cette fonction permet de construire un message UPDATE annulant des routes.

Pour pouvoir insérer plusieurs préfixes dans le même message sans connaître tous les préfixes avant l'appel de cette fonction, il faut conserver une trace du message en construction. Comme un message UPDATE est construit en fonction d'un certain pair BGP, il suffit de mettre dans la structure gardant les informations de celui-ci, un champ stream supplémentaire qui représente le message UPDATE en construction permettant d'annuler des routes.

Comme on peut le constater sur la figure 4.1, l'envoi du message en construction est effectuée lorsqu'une des deux conditions suivantes survient :

1. le message en construction a atteint sa taille maximale qui est de 4096 octets,
2. il n'y a plus de routes dans le message UPDATE reçu qui a initié la composition du message actuel. Autrement dit, le processus de décision activé par le message UPDATE reçu est terminé.

Les deux conditions explicitées précédemment s'appliquent lors de la réception d'un UPDATE. Cependant, il existe d'autres situations où un message UPDDATE contenant des routes à annuler peut être construit, comme la rupture de session avec un pair. Dans

toutes ces situations, la deuxième condition est modifiée de telle manière qu'il s'agisse de la fin de la fonction déclenchée par cet événement.

void bgp_update_build (struct peer *peer, struct attr *attr, struct prefix *p) Cette fonction est la fonction principale par laquelle un message UPDATE va être construit. C'est cette fonction qui va différer dans les deux versions de la modification du comportement apporté à *BGPd*. Celle-ci est décrite dans les deux sections qui suivent.

4.2 Spécificités de la première version

La première version de la fonction de construction de messages UPDATE est la plus simple. Les messages UPDATE annonçant des routes sont construits incrémentalement et ils sont envoyés dès qu'une des conditions suivantes est rencontrée, comme on peut le voir sur la figure 4.2 :

1. la taille du message actuel + la taille du nouveau préfixe à insérer est strictement plus grand que 4096 octets,
2. les attributs de la route à annoncer, donc à insérer dans le message, ne sont pas les mêmes que ceux du message UPDATE en construction. Il est évident que cette condition ne peut survenir que lors d'une annonce et non d'une annulation,
3. toutes les routes d'un message UPDATE reçu ont été traitées.

Comme il a déjà été dit précédemment, il existe d'autres situations où un message UPDDATE contenant des routes à annuler peut être en construction, comme la rupture de session avec un pair. Dans toutes ces situations, la troisième condition se trouve être dès lors la fin de la fonction déclenchée par cet événement.

Ces conditions ne permettent donc pas de construire en même temps plusieurs messages UPDATE pour un pair. Par contre, on peut espérer mettre plusieurs préfixes dans un même message pour autant que les attributs ne changent pas à chaque insertion d'un préfixe dans le message.

Il doit, comme pour le cas d'un message UPDATE annulant des routes, exister une structure stream permettant de retenir le message en construction. Mais il faut aussi pouvoir déterminer quels sont les attributs de ce message. On pourrait très bien extraire les attributs du message à chaque fois que l'on insère une route dans le message pour comparer l'égalité des attributs des deux routes. Cependant, comme ces attributs sont quand même sauvegardés dans BGPd, il suffit de pointer vers la structure contenant ces attributs. On évite ainsi un travail CPU supplémentaire au détriment d'un minimum de mémoire supplémentaire (4 octets par pair BGP).

Une fois que toutes les routes d'un message UPDATE reçu ont été traitées, tous les messages ne sont pas forcément encore envoyés vers les pairs concernés. Donc, il faut, à la fin du traitement du NLRI d'un message UPDATE reçu, appeler une fonction qui va envoyer le message en construction pour chaque pair. Ceci s'effectue en appelant pour chaque pair la fonction ***void bgp_packet_update_clear(struct peer *peer)***.

Commentaires Plusieurs avantages existent avec cette implémentation. Il n'y a pas trop de mémoire gaspillée pour pouvoir la supporter, 8 octets supplémentaires par pair par rapport à l'implémentation existante. Le délai avant que le message ne soit mis dans la file FIFO ne dépasse pas le temps de traitement d'un UPDATE.

Par contre, ce délai est évidemment plus grand que celui de l'implémentation originale puisque celui-ci insérait le message UPDATE après chaque route traitée. Un inconvénient

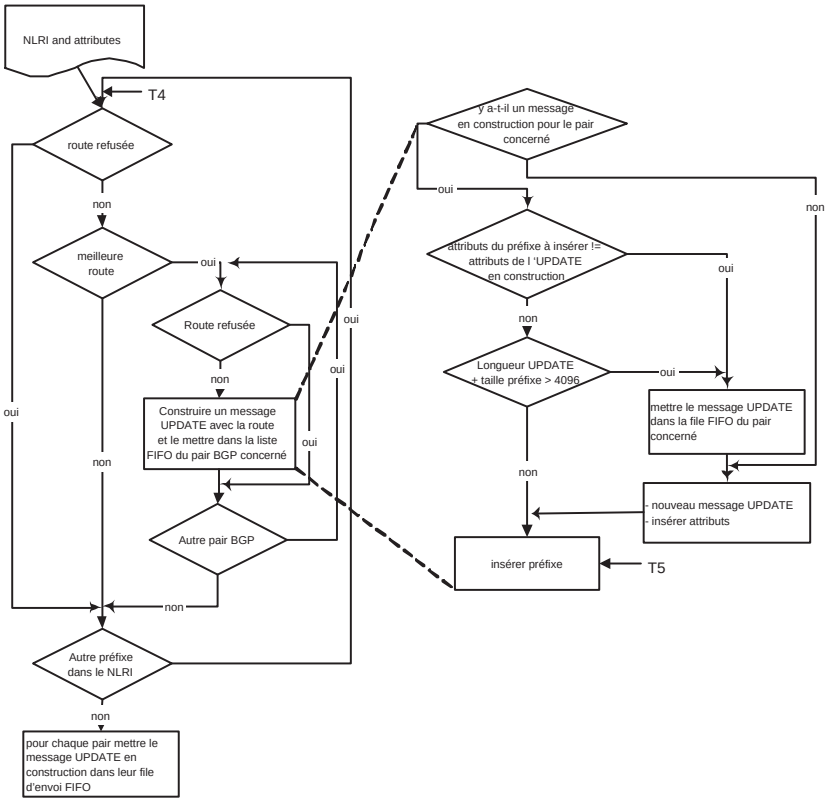


FIG. 4.2 – *Changement pour l'envoi des UPDATE - première version*

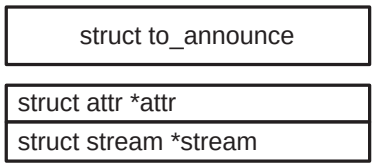


FIG. 4.3 – *Structure de la première solution de bgp_update_build*

à cette implémentation-ci est qu'elle n'essaye pas d'optimiser avec efficience la taille des messages UPDATE. En effet, si chaque route passant par le processus de décision doit être annoncée avec des attributs différents pour chacune, alors BGPd envoie des messages UPDATE ne contenant à nouveau qu'une seule route [41].

4.3 Spécificités de la deuxième version

La deuxième version de la fonction de construction de messages UPDATE annonçant des routes essaye d'optimiser la taille d'un message UPDATE avec certaines limites. La solution consiste à n'envoyer un message UPDATE que lorsqu'il est complet. En fait, le changement principal réside dans le fait qu'un changement d'attribut n'implique pas l'envoi d'un message UPDATE. Donc, un message UPDATE est envoyé si une des conditions suivantes est rencontrée, comme on peut le voir sur la figure 4.4 :

[41] Ceci est purement théorique et ne devrait pas arriver en pratique.

1. la taille du message + la taille du nouveau préfixe à insérer est strictement plus grand que 4096 octets,
2. toutes les routes d'un message UPDATE reçu ont été traitées.

Comme pour la première version, il existe d'autres situations où un message UPDATE contenant des routes à annuler peut être en construction. Dans toutes ces situations, la deuxième condition est modifiée pour devenir la fin de la fonction déclenchée par l'événement.

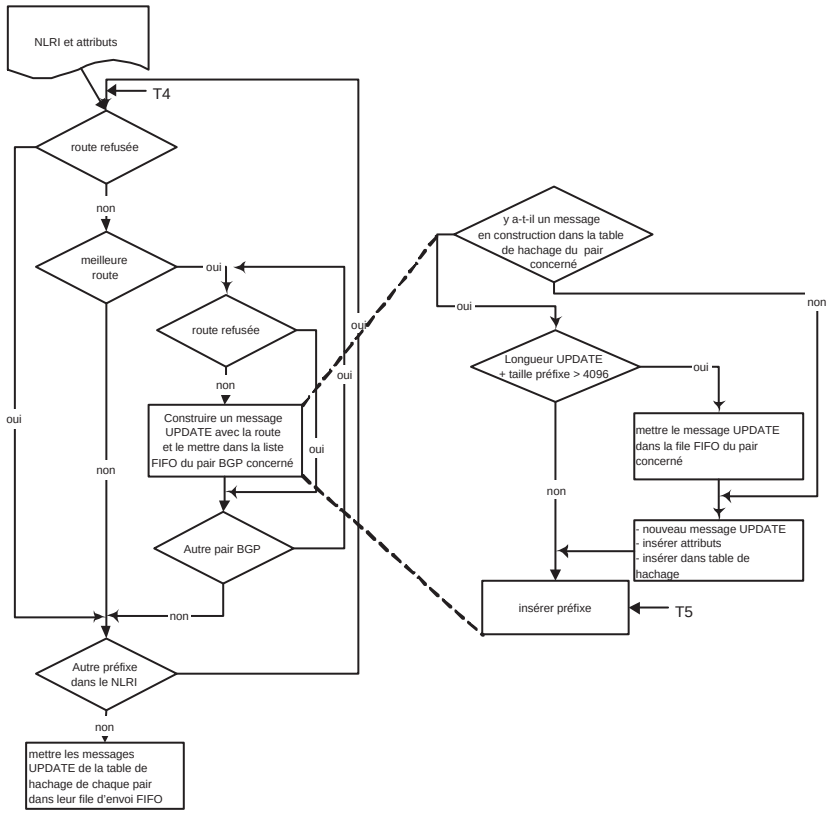


FIG. 4.4 – *Changement pour l'envoi des UPDATE - deuxième version*

Avec ces deux conditions, il est tout à fait possible de construire plusieurs messages UPDATE en même temps. Il faut donc une structure retenant plus d'informations que pour la première version de la solution. En fait, il faut pouvoir retenir plusieurs fois la structure to_announce. Pour ce faire, une table de hachage est utilisée, comme montré sur la figure 4.5. Elle est utilisée dans deux cas :

1. lorsque l'on doit retenir une nouvelle structure to_announce c'est-à-dire quand on n'a pas encore créé de messages UDPATE avec les attributs de la route que l'on veut annoncer,
2. lorsque l'on veut accéder à une structure to_announce contenant un message UPDATE dont les attributs sont les mêmes que ceux de la route que l'on veut annoncer.

La clé de cette table de hachage est basée sur les valeurs des attributs. Il existe une fonction dans BGPd qui crée une telle clé. Malheureusement, comme BGPd inclut l'attribut *WEIGHT* dans la structure attr, une fonction similaire a dû être réécrite pour ne plus tenir

compte de cet attribut qui n'est qu'un attribut local à un routeur.

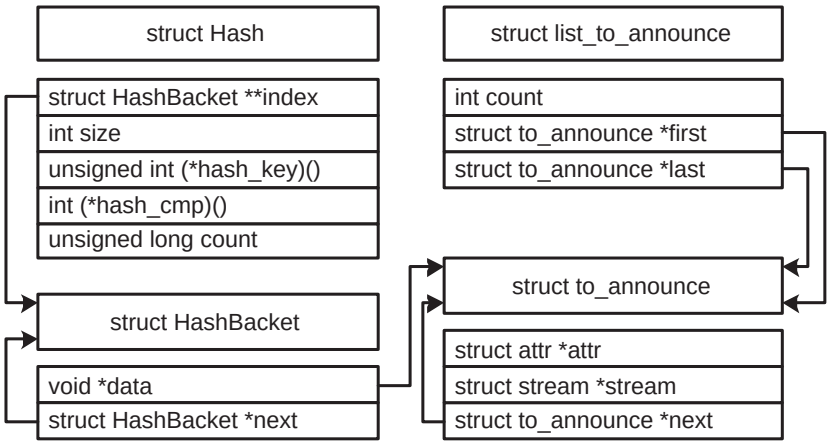


FIG. 4.5 – *Structure de la deuxième solution*

Sur la figure 4.5, on peut voir qu'il existe une structure appelée *list_ to_ announce*. Cette structure est utilisée lorsque toutes les routes du message UPDATE reçu ont été traitées. En effet, lorsque toutes ces routes ont été traitées, il peut rester des messages dans la table de hachage qui n'ont pas encore été envoyés car ils n'ont pas atteint leur taille maximale. Donc, il faut pouvoir envoyer tous ces paquets. Pour ce faire, la table de hachage n'est pas utilisée car ce n'est pas une structure efficiente pour le parcours séquentiel des données. Donc, en même temps que l'on insère un élément dans la table de hachage, on crée une liste chaînée qui permet de faire ce parcours de manière plus efficiente.

struct Hash			
Champ	**Quantité**	**Taille**	**Total**
index	1	4 octets	4
**index*	2048	4 octets	8192
size	1	4 octets	4
hash_ key	1	4 octets	4
hash_ cmp	1	4 octets	4
count	1	4 octets	4
struct list_ to_ announce			
Champ	**Quantité**	**Taille**	**Total**
count	1	4 octets	4
next	1	4 octets	4
last	1	4 octets	4
Total			8224 octets

FIG. 4.6 – *Taille de la table de hachage vide initialisée*

Commentaires Comme on peut le voir en détail sur la figure 4.6, la structure à vide prend déjà 8224 octets et ce pour chaque pair. La figure 4.7 montre qu'il faut rajouter

4112 octets pour chaque message contenu dans la table de hachage. Ce qui fait qu'à tout moment la taille, en octets, de la table de hachage est égale à $8224 + n * 4112$ où n est le nombre de messages en construction.

struct HashBacket	
Champ	**Taille**
data	4 octets
next	4 octets
struct to_announce	
Champ	**Taille**
attr	4 octets
stream	4096 octets
next	4 octets
Total	4112 octets

FIG. 4.7 – *Taille supplémentaire lors du rajout d'un message*

Le délai maximum avant l'envoi du message devrait théoriquement être égal au temps pris pour traiter l'entièreté du paquet plus le temps mis pour mettre les derniers paquets dans la file d'attente FIFO lors de la réception d'UPDATE.

Cette implémentation a l'avantage sur la version précédente qu'elle essaye d'optimiser la taille d'un message UPDATE puisqu'elle n'envoie pas automatiquement le message UPDATE lors d'un changement d'attribut entre deux routes consécutives que l'on veut annoncer à un pair BGP.

4.4 Tests sur l'implémentation

Cette section présente les mêmes tests que ceux effectués pour l'implémentation originale de BGPd. Comme deux modifications différentes ont été apportées, deux séries de tests sont présentées. Les deux séries sont présentées conjointement. Ceci permet une comparaison plus directe entre les deux solutions. En ce qui concerne, la comparaison avec l'implémentation originale, il n'en est souligné que les points forts. Le soin est laissé au lecteur d'effectuer une comparaison plus poussée en se reportant aux sections décrivant les résultats des tests de l'implémentation originale.

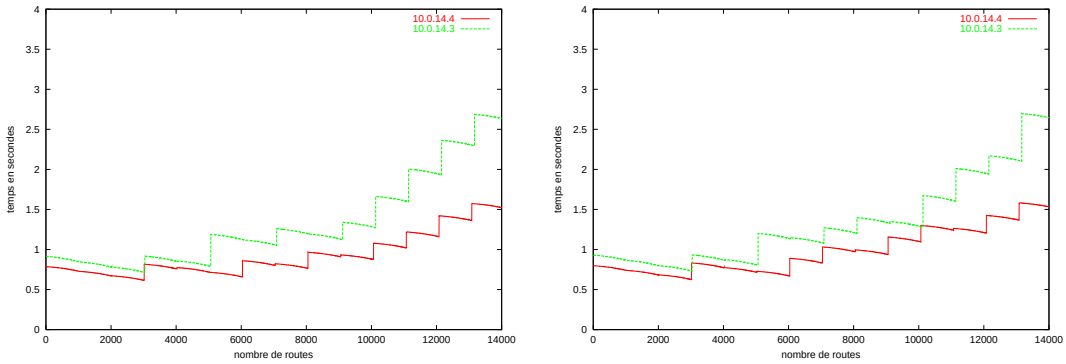
Matériel utilisé Le matériel utilisé est le même que celui décrit à la section 3.3.1 à la page 40.

4.4.1 Synchronisation de la table de routage

Les deux tests sur la synchronisation de la table de routage ont le même testbed et déroulement que ceux décrits en section 3.3.2 page 41.

Résultats Les figures 4.8(a) et 4.8(b) représentent le temps global de synchronisation de la table de routage pour, respectivement, la première version et la deuxième version de la modification apportée. Les deux mesures prises pendant ce test sont :

1. T1 : le moment où *BGPd* trouve une route à envoyer
2. T2 : le moment de l'envoi du message UPDATE



(a) *Première solution* (b) *Deuxième solution*

FIG. 4.8 – *Temps d'envoi de la table de routage*

Première version Le temps total d'envoi de la table de routage à destination du RR-client est 1,52 secondes et de 2,64 pour le pair eBGP.

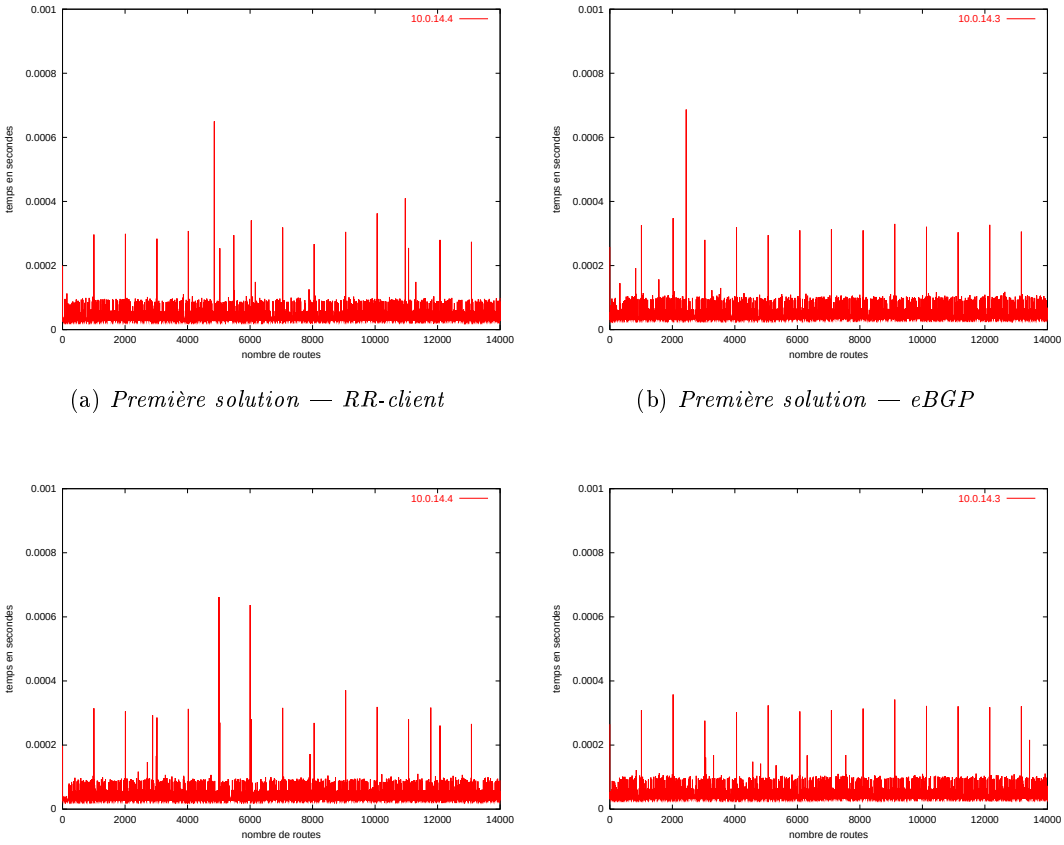
Deuxième version Le temps total d'envoi de la table de routage à destination du RR-client est de 1,53 secondes et de 2,645 pour le pair eBGP.

Comparaison Les tests menés sur les deux versions donnent des résultats semblables. Par contre, en comparant avec la version originale de *BGPd*, on peut remarquer que les temps sont beaucoup plus petits pour l'implémentation modifiée. Il faut plus de 10 secondes à l'implémentation originale pour annoncer toute sa table de routage. La raison en est simple, au lieu d'envoyer 14000 messages UPDATE, *BGPd* n'envoie plus que quatorze de ces messages. Le temps le plus important étant le temps qu'il faut pour parcourir la table de routage.

Les figures 4.9 représentent le temps mis par *BGPd* pour construire tous les messages UPDATE pour chacun des pairs BGP. Les deux prises de temps sont :

1. T1 : le moment où *BGPd* trouve une route à envoyer
2. T2 : le moment où le préfixe est inséré dans le message. Le moment du time-stamp diffère par rapport au premier test pour une raison assez simple. La fonction dans laquelle se retrouvait ce time-stamp est celle qui a été modifiée. Dès lors, on prend comme point de référence la fin de cette fonction modifiée qui correspond ici à l'insertion du préfixe dans le message UPDATE.

Première version Sur les figures 4.9(a) et 4.9(b), le temps moyen mis pour traiter une route est de $3,288\ 10^{-5}$ seconde pour le route-reflector client et de $3,834\ 10^{-5}$ pour le pair eBGP. Les temps totaux sont de 0,46 seconde pour le RR-client et de 0,54 pour le pair eBGP. Les figures 4.10(a) et 4.10(b), représentant les distributions des figures 4.9(a) et 4.9(b), indiquent que le temps d'insertion du préfixe est régulier. Cela se confirme par l'écart-type du RR-client qui est de $1,32\ 10^{-7}$ et pour le pair eBGP, de $1,36\ 10^{-7}$.



(a) *Première solution — RR-client* (b) *Première solution — eBGP*

(c) *Deuxième solution — RR-client* (d) *Deuxième solution — eBGP*

FIG. 4.9 – *Temps de construction des messages UPDATE*

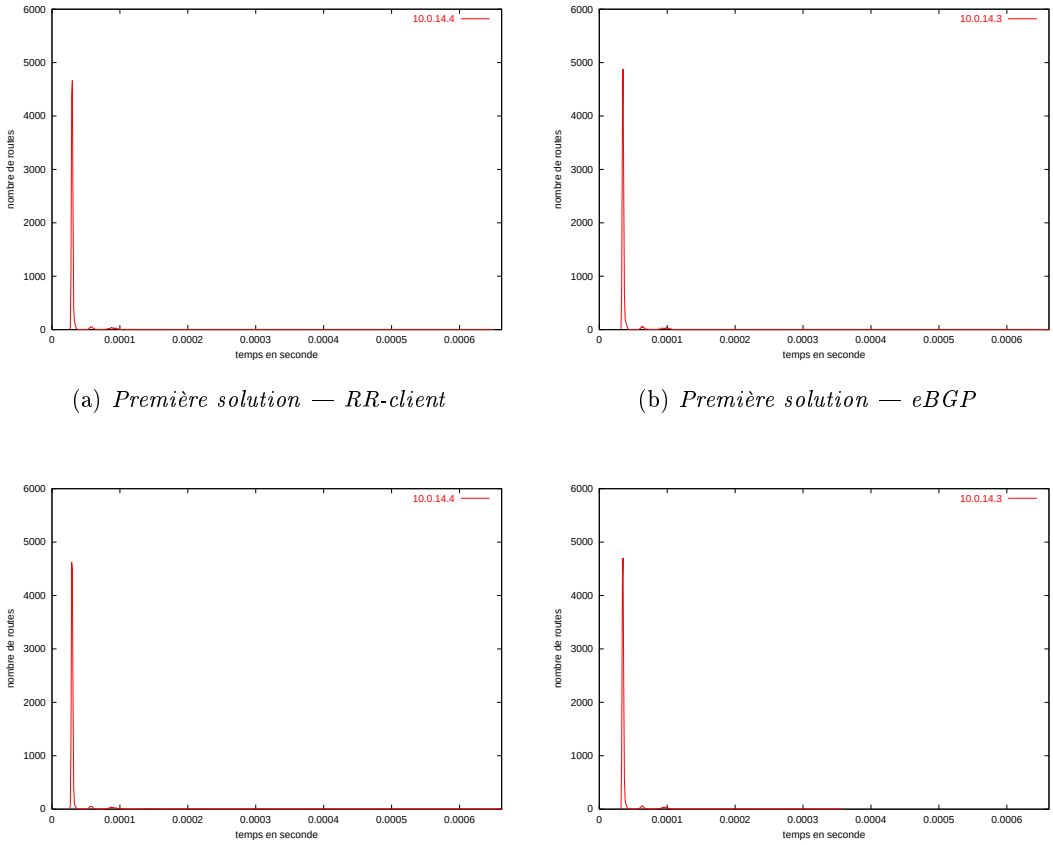
Deuxième version Sur les figures 4.9(c) et 4.9(d), le temps moyen est de $3,281\ 10^{-5}$ seconde pour le RR-client et il est de $3,79\ 10^{-5}$ pour le pair eBGP. Les temps totaux sont de 0,46 seconde pour le RR-client et de 0,53 pour le pair eBGP. De même que pour la première version, leur distribution est très rapprochée (figures 4.10(c) et 4.10(d)) et est confirmé par leur écart-type qui est de $1,38\ 10^{-7}$ pour le RR-client et de $1,25\ 10^{-7}$ pour le pair eBGP.

Comparaison On peut remarquer sur les figures 4.9(a), 4.9(b), 4.9(c), 4.9(d) que *BGPd* met les paquets dans la file d'attente lors des pics qui se trouvent aux alentours des multiples de 1000 sur l'axe des abscisses.

Les temps sont comparables entre les deux versions de la modification apportée à *BGPd*. Par contre ceux-ci sont environ deux fois plus élevés en ce qui concerne l'implémentation originale.

4.4.2 Traitement des UPDATE par BGPd

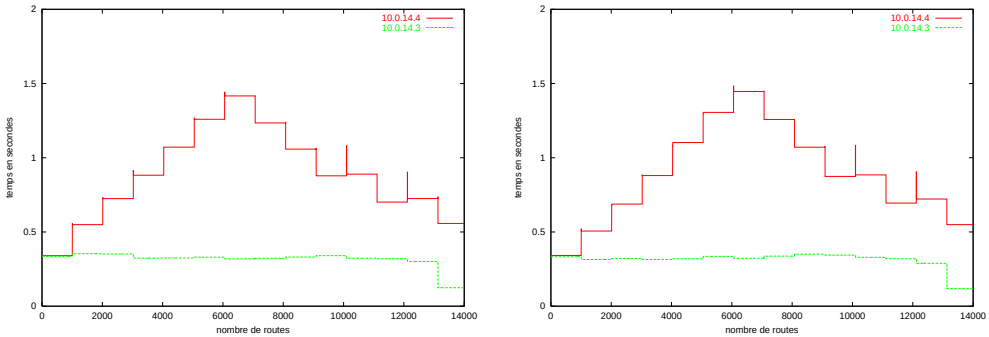
Les trois tests sur le traitement des messages UPDATE par *BGPd* ont le même testbed et déroulement que ceux décrits en section 3.3.3 page 44.



(a) *Première solution — RR-client* (b) *Première solution — eBGP*

(c) *Deuxième solution — RR-client* (d) *Deuxième solution — eBGP*

FIG. 4.10 – *Distribution du temps de construction des messages UPDATE*



(a) *Première solution* (b) *Deuxième solution*

FIG. 4.11 – *Temps de traitement global*

Résultats Les figures 4.11(a) et 4.11(b) représentent le temps de traitement global d'une suite de messages UPDATE par *BGPd*. Les deux mesures prises pour ce test sont comme indiqué sur la figure 3.7 page 45 :

1. T1 : l'arrivée d'un paquet contenant N routes
2. T2 : le départ d'un message UPDATE à destination d'un des deux routeurs Cisco contenant une route.

On obtient donc j time-stamps T2 correspondant au nombre de messages UPDATE à réannoncer aux pairs BGP et i time-stamps T1 correspondant aux messages UPDATE reçus. L'équation par laquelle on trouve le temps de traitement global d'un UPDATE est la suivante : $T2_j - T1_i$ (où i est le numéro du message UPDATE dans lequel se trouve les routes du message j).

Première version Le temps total mis pour envoyer tous les messages est de 3,05914 secondes pour le RR et de 2,63087 pour le pair eBGP. Le temps moyen entre la réception d'une route et son envoi est de 0,88128 pour le RR et de 0,31628 seconde pour le pair eBGP.

Deuxième version Le temps total mis pour envoyer tous les messages est de 3,04417 secondes pour le RR et de 2,61794 seconde pour le pair eBGP. Le temps moyen entre la réception d'une route et son envoi est de 0,88394 seconde pour le RR et de 0,3126 pour le pair eBGP.

Comparaison Les temps sont comparables comme on pouvait s'y attendre entre les deux versions de la modification de comportement de *BGPd*. Par contre, en ce qui concerne la comparaison avec l'implémentation originale, il est certain que les deux versions supplantent largement l'implémentation originale qui elle a besoin d'une dizaine de secondes pour envoyer tous les messages UPDATE. A nouveau pour la simple raison que *BGPd* n'envoie plus qu'entre 14 et 28 messages au lieu de 14000.

La raison pour laquelle le graphe est en *"escalier"* en ce qui concerne le réflecteur de route client est assez complexe. Premièrement, lorsque BGPd doit faire suivre une route vers ce client, il rajoute deux attributs. Le premier attribut rajouté est l'attribut ORIGINATOR-ID et le deuxième est le CLUSTER-LIST. Ce qui entraîne l'envoi de deux messages UPDATE pour la réception d'un message UPDATE de taille quasiment maximale comme ceux reçus lors des tests.

Deuxièmement, en observant le comportement de BGPd lors de ces phases de tests, il a été remarqué que BGPd réceptionne d'abord deux messages UPDATE avant tout envoi. Ceci implique qu'avant l'envoi du premier message, il y a déjà le premier message UPDATE reçu qui a été traité, ce qui est tout à fait normal, mais le deuxième message a aussi été traité. A partir de ce moment là, BGPd va envoyer deux messages, un vers le routeur 10.0.14.3 et l'autre vers le routeur 10.0.14.4. Une fois qu'il a effectué cette séquence, il continue toujours la même séquence suivante :

1. traitement d'un message UPDATE reçu,
2. envoi d'un message UPDATE vers les deux routeurs.

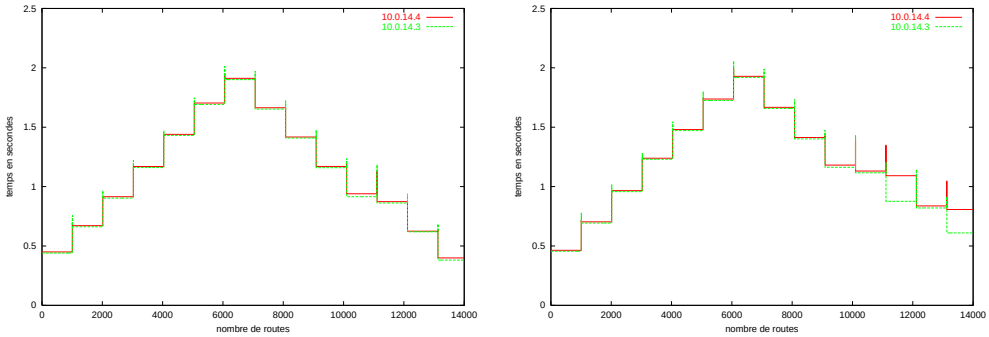
Une fois qu'il n'y a plus de messages UPDATE à traiter, *BGPd* envoie bien entendu le reste des messages UPDATE restants.

Ce comportement explique donc pourquoi BGPd envoie les messages du route reflector client avec un délai plus long que pour ceux du pair eBGP. En effet, à chaque fois qu'un message UPDATE est traité, BGPd envoie un message vers les routeurs, or il y a deux messages à envoyer au route reflector client. La phase descendante du graphe concernant le

route reflector client commence lorsque BGPd n'a plus de message UPDATE en réception à traiter.

Ce comportement explique aussi le temps qu'il faut pour envoyer le premier paquet qui est égal à deux fois le temps qu'il faut pour traiter deux messages UPDATE. Quant au dernier paquet envoyé vers le pair eBGP, il lui faut moins de temps car il n'y a eu comme délai que le temps de traitement du message UPDATE réceptionné contenant les routes qu'il contient et l'envoi de deux messages UPDATE.

Il reste cependant encore certains points d'ombres inexpliqués dans les variations vues dans le tracé du route reflector client.



(a) *Première solution* (b) *Deuxième solution*

FIG. 4.12 – *Illustration du comportement de la politique d'envoi de BGPd*

Pour nous convaincre de ce raisonnement, le même test a été effectué à une différence près, BGPd a été configuré de sorte que celui-ci ajoute plusieurs fois son numéro d'AS dans l'AS-PATH d'une annonce à faire au pair eBGP. Cette configuration fait qu'à chaque réception d'un message UPDATE, BGPd doit envoyer deux messages à chacun des pairs et non plus seulement au pair iBGP. Les figures 4.12(a) et 4.12(b) représentent bien l'effet d'escalier attendu.

De même sur la figure 3.8 à la page 46, on peut croire qu'il y a un grand temps de latence entre la réception du premier message UPDATE et le premier envoi. Pourtant si nous effectuons un zoom sur la première partie du test comme sur la figure 4.13, on peut constater le même comportement. En effet, le premier message est envoyé après quelque 0,369 seconde en direction de chaque pair. Ce qui équivaut à peu de choses près à deux fois le traitement de deux messages UPDATE de la part de *BGPd*. De plus, on constate qu'après 2,545 secondes, l'envoi devient plus fréquent ce qui correspond au temps de traitement de tous les UPDATE reçus par *BGPd*.

Les figures 4.14(a) et 4.14(b) représentent le temps de traitement des routes cumulé par paquets. Ce test-ci prend deux time-stamps, T1 et T3 indiqué sur la figure 3.7 page 45 :

1. T1 : l'arrivée d'un paquet contenant N routes
2. T3 : le moment de la construction du message UPDATE pour une route à envoyer vers un autre routeur. Une différence existe entre la prise de temps de l'implémentation originale et les deux versions de la modification apportée. Ce moment correspond plus précisément à l'insertion du dernier préfixe dans le message UPDATE.

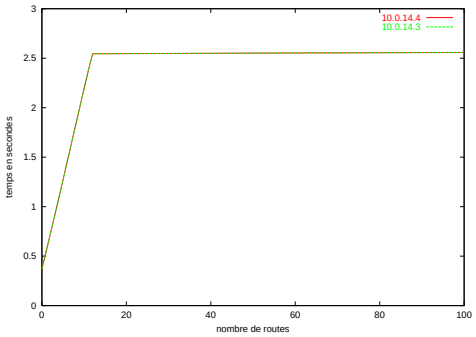
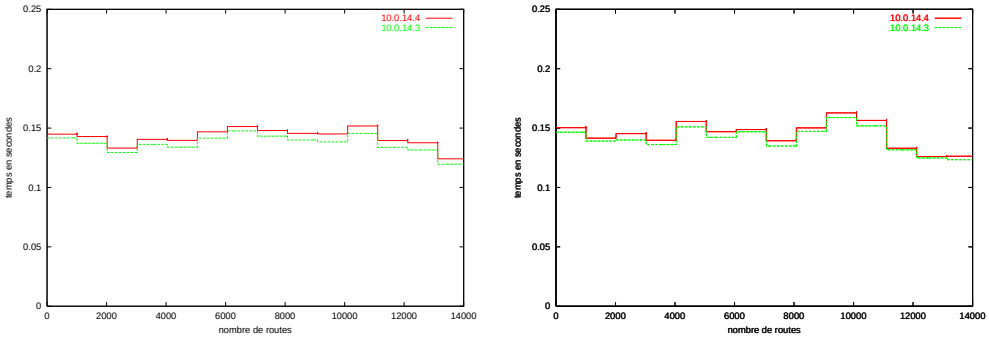


FIG. 4.13 – *Zoom sur le traitement global de l'implémentation originale*



(a) *Première version* (b) *Deuxième version*

FIG. 4.14 – *Temps de traitement des routes*

Nous obtenons donc j time-stamps T3 correspondant au nombre de messages UPDATE à annoncer aux pairs BGP et i time-stamps T1 correspondant aux messages UPDATE reçus. L'équation par laquelle nous trouvons le temps de traitement est la suivante : $T3_j - T1_i$ (où i est le numéro du message dans lequel se trouve les routes du message j).

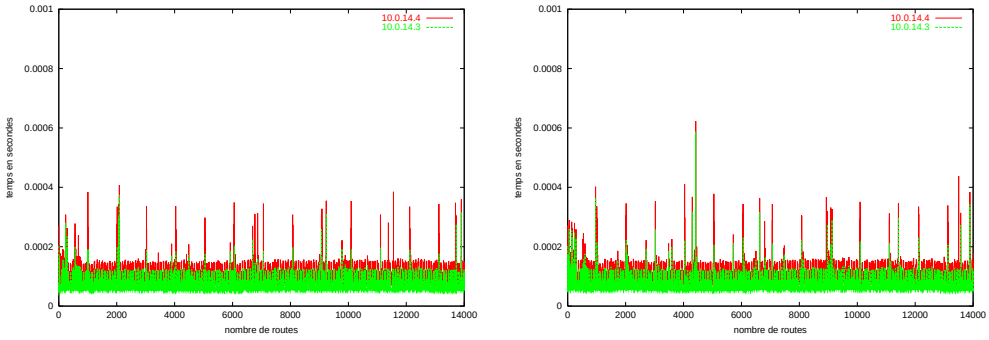
Première version Le temps moyen de construction d'un message UPDATE est de 0,142 seconde pour le RR-client et de 0,137 seconde pour le pair eBGP.

Deuxième version Le temps moyen est d'environ 0,145 seconde pour le RR-client et de 0,141 pour le pair eBGP.

Comparaison On peut remarquer qu'il faut envoyer deux messages UPDATE par message UPDATE reçu pour le RR-client par les petites excroissances ressortant du tracé. Ces excroissances s'expliquent par le fait qu'il faille d'abord finaliser le message et le mettre dans le file d'envoi FIFO du pair avant de construire un autre message.

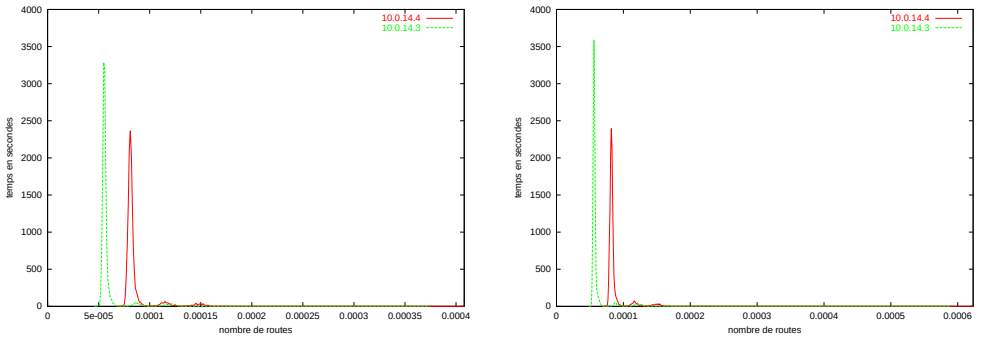
A nouveau, les temps sont comparables pour les deux versions de la modification apportée mais légèrement supérieure pour la deuxième version. Et, ceci en raison d'un ajout plus conséquent dans le code. La physionomie du graphe a changé entre les deux versions

de la modification et l'implémentation originale. Il est tout à fait normal de voir apparaître les lignes horizontales, puisque celles-ci correspondent au temps mis pour construire le message UPDATE. Dans la première version, tous les messages ne sont constitués que d'un seul préfixe. Donc, le temps est croissant au fur et à mesure que l'on traite le message UPDATE reçu. Par contre, dans les implémentations modifiées, un délai supplémentaire est forcément introduit pour pouvoir construire le message avec plusieurs préfixes. De plus, on peut remarquer que le traitement du paquet demande plus de temps de la part de la version originale de l'implémentation.



(a) *Première version* (b) *Deuxième version*

FIG. 4.15 – *Temps de traitement jusqu'à la fin de construction du message*



(a) *Première version* (b) *Deuxième version*

FIG. 4.16 – *Distribution du temps de traitement jusqu'à la fin de construction du message*

Les figures 4.15(a) et 4.15(b) représentent le temps de traitement de chaque route jusqu'à la construction du message UPDATE. Les deux temps pris sont, comme indiqué sur la figure 4.2 page 53 pour la première version et sur la figure 4.4 page 54 pour la seconde version :

1. T4 : le début du traitement d'une route
2. T5 : la fin de la construction du message UPDATE pour une route à envoyer vers un autre routeur. Il faut remarquer que la fin de la construction du message n'est pas

lorsque celui-ci est finalisé et mis dans la file d'envoi du pair BGP mais lorsque le préfixe est inséré dans le message UPDATE.

Première version Le temps moyen est de $8,6\ 10^{-5}$ seconde pour le RR-client et de $5,9\ 10^{-5}$ pour le pair eBGP. Le maximum est de 0,000408 seconde et est probablement dû à une prise de contrôle de l'OS. Cependant, la distribution (figure 4.16(a)) est régulière et l'écart-type est de $1,64\ 10^{-7}$ pour le RR-client et de $1,28\ 10^{-7}$.

Deuxième version Le temps moyen pour la figure 4.15(b) est de $8,7\ 10^{-5}$ seconde pour le RR-client et de $6,0\ 10^{-5}$ pour le pair eBGP. A nouveau, la distribution (figure 4.16(b)) est régulière ce qui se confirme par l'écart-type de $1,7\ 10^{-7}$ pour le RR-client et de $1,4\ 10^{-7}$ pour le pair eBGP.

Comparaison Les temps sont à nouveau comparables entre les deux versions du changement de comportement de *BGPd* bien que très légèrement supérieur pour la deuxième version. Et en ce qui concerne la comparaison avec l'implémentation originale, bien qu'il y ait du code en plus, ces temps sont moindres.

4.4.3 Premières Conclusions

En comparant les tests menés pour les trois versions de *BGPd*, on peut aisément remarquer que les deux versions modifiées améliorent largement les performances de *BGPd*. La première version a l'air de perdre légèrement moins de temps en traitement. Cependant, pour se donner une idée plus précise du véritable comportement des deux versions, des tests plus approfondis sont menés dans la prochaine section.

4.5 Tests approfondis

Des tests approfondis ont été effectués pour montrer le comportement de la version originale et des deux versions modifiées de *BGPd* dans des conditions réelles. Les tests menés sont les mêmes à l'exception des données injectées dans les tests. En effet, dans les séries de tests présentées précédemment, les routes avaient toutes les mêmes attributs, ce qui n'arrive jamais dans des conditions réelles d'utilisation. Donc, les tests suivants présentent les performances avec des routes dont les attributs sont réels. Cependant, deux tests ne s'y trouvent pas car ils n'apportent aucune information supplémentaire. Il s'agit du test représentant le temps de traitement de chaque route sans prendre en compte l'envoi lors de l'ouverture d'une session et du test équivalent lors de la réception des messages UPDATE.

De plus, pour permettre de mettre à l'épreuve les implémentations, deux séries de tests ont été effectués pour chaque type de tests. Le premier injecte des préfixes avec peu d'attributs différents et le deuxième n'a que peu d'attributs identiques. Ces deux séries vont permettre de se rendre compte de l'influence des attributs sur les temps de traitement.

4.5.1 Matériel utilisé

Le matériel et logiciel utilisés pour effectuer ces tests a quelque peu changé :
- 4 PC Intel Celeron 300 MHz avec comme OS Linux noyau 2.4.18
  L'un d'eux fait tourner *BGPd*, deux autres font tourner sbgp et le dernier MRTd [42].

[42] Disponible à l'adresse http ://www.merit.edu/ mrt.

- 2 routeurs Cisco 3600 (version d'IOS 12.2(7)) avec 64 MB de mémoire vive et 6 interfaces Ethernet/IEEE 802.3.
- sbgp[oMN01] : sbgp permet de jouer une session BGP à partir d'un fichier enregistré au format binaire de MRT.
- MRTd : c'est une implémentation du protocole de routage BGP-4. Celle-ci va nous permettre de charger une table de routage au format binaire MRT.

Grâce à l'outil d'Olaf Maennel, RTG, on a pu disposer d'une table de routage réaliste et des messages UPDATE aussi. Ces messages UPDATE n'annulent aucune route, ils ne font qu'annoncer les mêmes routes que celles qui se retrouvent dans la table de routage. Il y a 21821 routes générées par RTG. Le premier test consiste à avoir 1641 attributs différents pour ces 21821 routes. Tandis que le deuxième test compte 16877 attributs différents.

4.5.2 Synchronisation de la table de routage

Testbed La figure 4.17 montre que pour ce test, il a fallu deux PC, et les deux routeurs Cisco. Le premier PC fait tourner sbgp dont l'utilité ici est de charger la table de routage pour *BGPd*, le deuxième PC fait tourner *BGPd*.

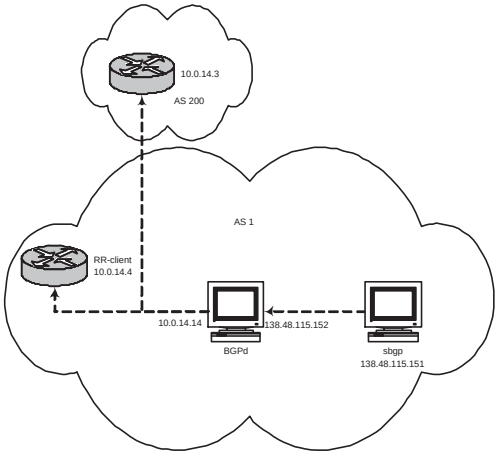
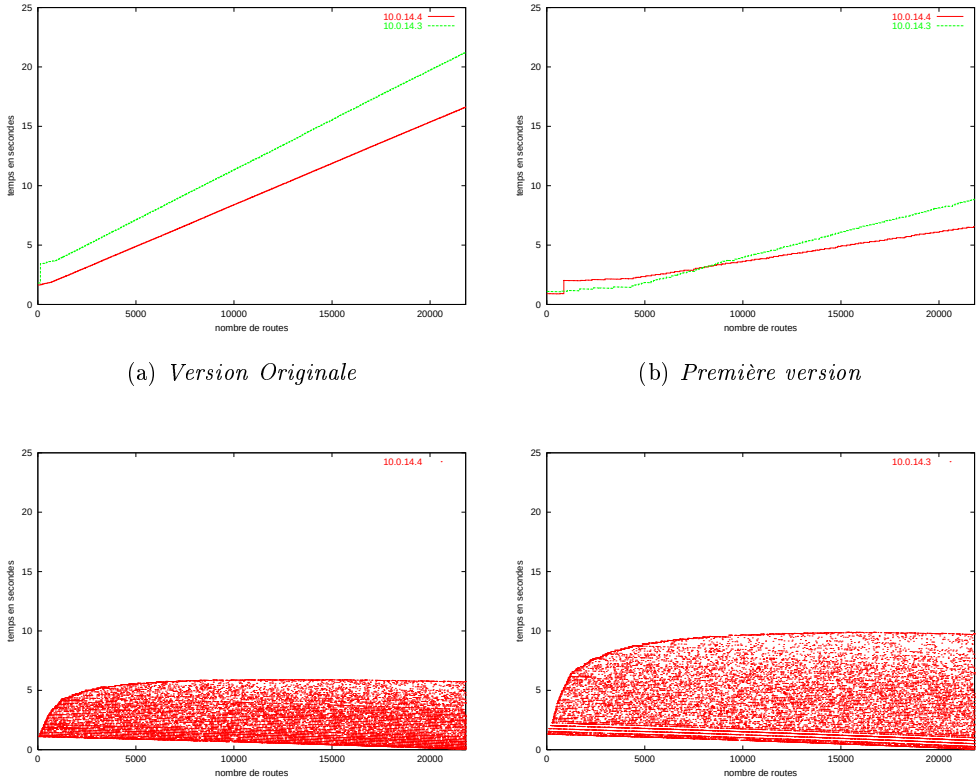


FIG. 4.17 – *Testbed pour la synchronisation de la table de routage*

Déroulement Au départ, on charge la table de routage via le PC qui fait tourner sbgp. Une fois cette table de routage chargée, on va pouvoir faire débuter la conversation entre *BGPd* et les routeurs Cisco. Pour pouvoir tester le temps de synchronisation, il faut que BGPd soit configuré de telle manière que lorsqu'il recevra sa table de routage, il ne la communique pas immédiatement au routeur Cisco, auquel cas nous prendrions des mesures sur le flooding de messages UPDATE reçus. Donc, lors du lancement de *BGPd*, les routeurs Cisco ne sont pas dans sa configuration. Une fois que BGPd aura toute sa LOC-RIB initialisée, on ajoutera à la configuration de *BGPd*, les routeurs Cisco. La configuration des routeurs Cisco est des plus élémentaires, puisque ceux-ci ne sont présents que pour initier une communication avec *BGPd*. Cette configuration ne comporte donc que la déclaration du pair BGP représenté par le PC faisant tourner *BGPd*. Dès que la synchronisation commence, quand la connexion passe dans l'état *Established*, le test commence. Une fois que le dernier message UPDATE est envoyé vers un des deux routeurs Cisco, le test s'arrête.

Résultats Le test reste le même que celui présenté à la page 56 pour les deux versions de la modification de comportement de *BGPd* et à la page 42 pour la version originale.



(a) *Version Originale*

(b) *Première version*

(c) *Deuxième version — RR-client*

(d) *Deuxième version — eBGP*

FIG. 4.18 – *Temps de synchronisation de la table de routage*

Version Originale Lors du premier test, figure 4.18(a) ne contenant que 1641 attributs différents, *BGPd* a mis 23,05 secondes pour envoyer toute sa table de routage vers le pair eBGP et 18,26 secondes pour l'envoyer vers le RR-client.

Lors du deuxième test contenant 16877 attributs différents, les temps restent les mêmes. En effet, cela n'a aucune influence sur le comportement de l'implémentation originale puisque le fait d'avoir plus de préfixes n'influencent aucunement le processus de construction des messages lors d'une ouverture de session dans l'implémentation originale.

Première Version La figure 4.18(b) montre qu'il a fallu moins de temps pour envoyer les messages UPDATE puisqu'il n'y en a que 1641 à envoyer. *BGPd* a donc mis 9,96 secondes pour envoyer sa table de routage vers le pair eBGP et 7,47 secondes pour le faire vers le RR-client.

Lors du deuxième test, figure 4.19, il a forcément fallu plus de temps puisque *BGPd* a envoyé 20896 messages UPDATE. Pour envoyer ces messages, il a fallu 25,29 secondes pour le pair eBGP et 19,52 secondes pour le RR-client.

Deuxième Version Le premier test figure 4.18(c) et 4.18(d) représentent un nuage de points. Ceci est tout à fait normal, puisque cette implémentation cherche à maximiser la taille d'un message UPDATE et que le parcours de la table de routage BGP est effectué en fonction des préfixes et non des attributs. Ceci implique deux points expliquant le tracé en nuage :

- pour tout message A, B, si la construction de A a débuté au temps N et celui de B au temps N+1 cela n'implique pas que le message A est envoyé avant le message B.
- si on a $\{a_1, \ldots, a_n\}$, l'ensemble des préfixes constituant un message UPDATE et $\{b_1, \ldots, b_m\}$, l'ensemble des préfixes constituant la table de routage BGP alors $\forall k, l : 1 \leq l < n \; et \; 1 \leq k < m, \; a_k = b_l \not\Rightarrow a_{k+1} = b_{l+1}$.

Le temps mis pour envoyer toute la table de routage vers le pair eBGP est de 11,04 secondes et de 6,86 secondes pour le RR-client. Cette version n'a envoyé que 1641 messages au total pour envoyer toute la table de routage.

Le deuxième test n'a donné aucun résultat. Le test n'arrive jamais à terme car pendant que BGPd parcourt sa table de routage pour trouver les meilleures routes, aucun autre processus, interne à BGPd, ne peut prendre la main. Ce parcours prend malheureusement plus de 30 secondes. Ce qui signifie, que lorsque BGPd doit envoyer un KEEPALIVE, il ne le fait pas. La conséquence directe est donc une coupure de session entre BGPd et les pairs avec lesquels il communique. Il y a deux causes essentielles qui ralentissent BGPd :

- Il apparaît que la fonction de création de la clé de hashage n'est pas performante. Ce qui fait que lorsqu'il y a peu d'attributs différents, cela ne se ressent pas. Par contre, lors du test avec 16877 attributs différents, cela s'est fort ressenti. L'investigation de ce côté-là n'a pas pu être effectuée pour des raisons de temps. Mais il semblerait que l'on pourrait améliorer les performances de cette fonction, pour ne pas avoir trop de collisions.
- La table de hachage utilisée prend trop de places pour ce qu'elle doit faire. Dans le cadre du testbed en place, le PC faisant tourner BGPd n'a que 64Mo de mémoire vive. De plus, lors du parcours de la table de routage, chaque route doit être insérée dans un des messages de la table de hachage. Il y a 16877 attributs différents et aucun message n'est envoyé avant la fin du parcours de la table, La table de hachage va donc être constituée de 16877 messages. Or, chaque nouveau message en construction est créé avec une taille initiale à la taille maximale d'un message de BGP-4. Ce qui fait qu'avant l'envoi d'un seul message, la table de hachage a déjà une taille de 66,2 Mo rien que pour les messages en construction pour un pair. Ceci fait intervenir du swapping qui fait aussi perdre du temps à ce processus.

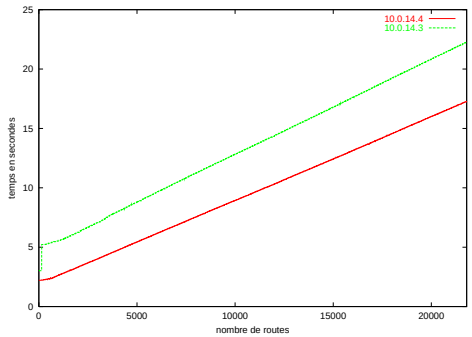


FIG. 4.19 – *Temps de synchronisation de la table de routage II*

Comparaison Après ces premiers tests, une conclusion peut déjà être tirée. On ne peut pas utiliser la deuxième version telle quelle pour au moins deux raisons :

1. La fonction de création de la clé de hachage n'est pas assez performante.
2. La quantité de mémoire vive demandée par la table de hachage lors d'une ouverture de session est beaucoup trop grande.

Si on veut pouvoir utiliser cette version lors d'ouverture de session, il faudrait donc modifier ces deux points.

Tandis qu'entre la première version de la modification et la version originale, tout dépend du rapport entre le nombre d'attributs sur le nombre de préfixe. Lorsque ce rapport est petit, la version modifiée prend légèrement plus de temps que la version originale tandis que si celui-ci est grand alors la version peut gagner énormément de temps vis-à-vis de la version originale. Donc, il semble que la première version de la modification soit viable même si celle-ci peut dégrader les performances de *BGPd* dans certaines conditions.

Néanmoins, il serait intéressant d'apporter une autre modification au comportement de *BGPd*. En effet, il n'y a pas moyen de parcourir la table de routage BGP en fonction des attributs sélectionnés mais seulement en fonction des préfixes. Ce changement de comportement pourrait peut-être améliorer ses performances, si on le couplait à la première version de la modification.

4.5.3 Traitement des UPDATE par BGPd

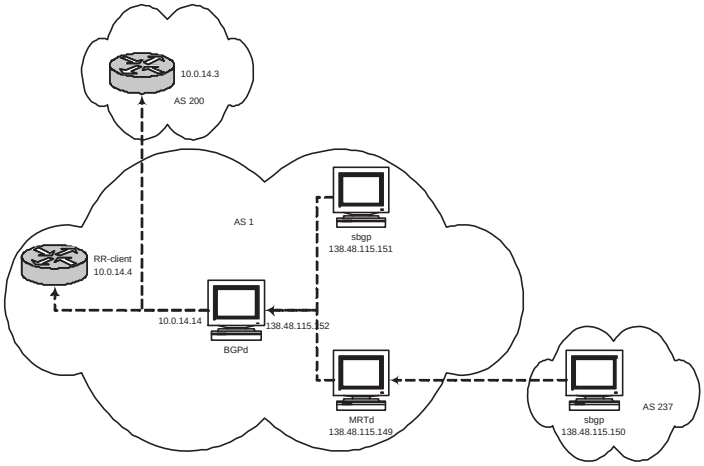
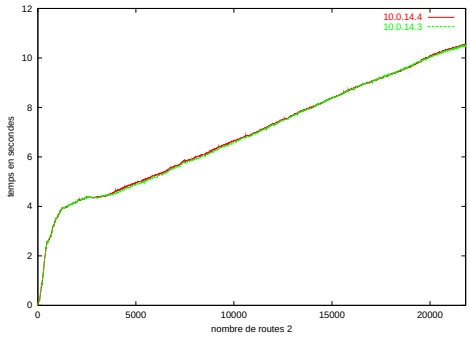


FIG. 4.20 – *Testbed pour les temps de traitement des messages UPDATE*

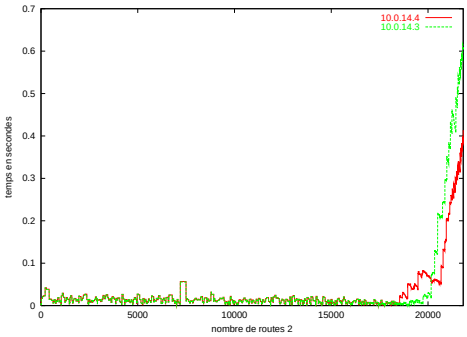
Testbed La figure 4.20 nous montre que le testbed est composé de quatre PC, et deux routeurs Cisco. Le premier PC fait tourner MRTd, et remplira la table de routage de *BGPd*. Le routeur de l'AS 237 envoie la route ayant comme attribut la communauté dont la sémantique est le début de test. Le troisième enverra des messages UPDATE via sbgp et c'est pour ces messages que les time-stamps sont pris. Et, enfin, le quatrième PC fait tourner *BGPd*. Les deux autres routeurs sont configurés pour communiquer avec *BGPd*, l'un est configuré comme Route Reflector-client et l'autre est un pair eBGP. On peut constater que les deux PC communiquant directement avec *BGPd* sont configurés pour être des pairs iBGP et non eBGP. De cette manière, BGPd n'essaie pas de redistribuer les routes de l'un vers l'autre pour éviter d'interférer avec l'évaluation du temps des traitements.

Déroulement La première étape de ce test consiste tout d'abord à remplir la table de routage de *BGPd*. Une fois la table de routage de *BGPd* remplie, le routeur de l'AS 237 envoie la route indiquant à BGPd le début de test. Ensuite, dès que la synchronisation de cette table avec les deux routeurs Cisco est finie, on peut commencer à envoyer des messages UPDATE vers *BGPd*. Ces messages UPDATE annoncent 21821 routes et 15623 de celle-si sont meilleures que celles qui constituent la table de routage. Le départ du test commence lors de la réception du premier message UPDATE et finit selon le cas, lors de l'envoi du dernier message UPDATE ou du traitement de la dernière route à envoyer.

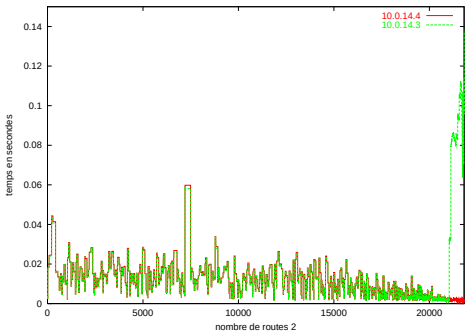
Il est à noter que lors du premier test, 15623 des 21821 routes sont de meilleures routes si elles viennent des messages UPDATE annoncés. Alors que dans le deuxième test, toutes les routes reçues sont meilleures.



(a) Version Originale



(b) Première version modifiée



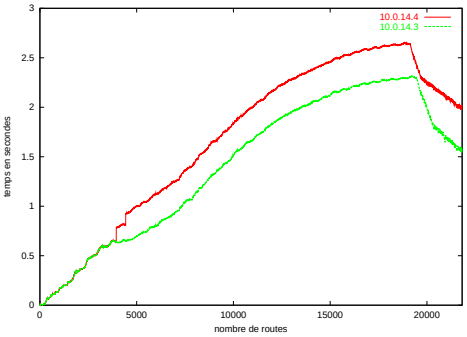
(c) Deuxième version modifiée

FIG. 4.21 – *Temps de traitement global*

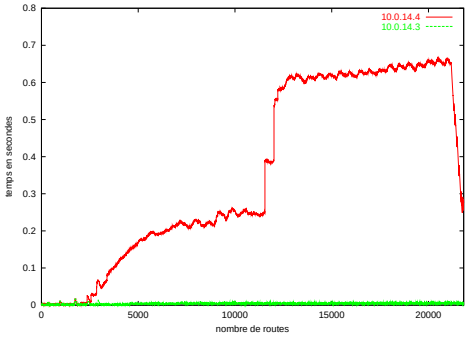
Résultats Les prises de temps de ce test sont les mêmes que ceux présentés page 45.

Version Originale Lors du premier test, figure 4.21(a), *BGPd* a mis 15,34 secondes pour appréhender tous les messages UPDATE et renvoyer les 15623 routes vers le pair eBGP. Il a fallu 15,41 secondes pour effectuer la même tâche pour le RR-client.

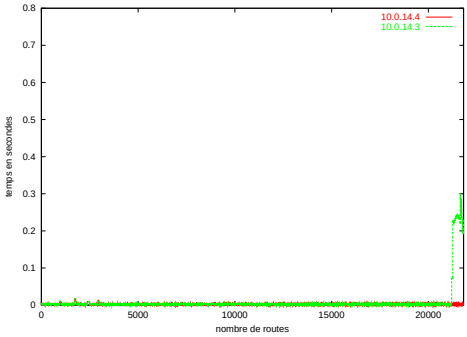
Le deuxième test, figure 4.22(a), quant à lui a pris un peu plus de temps puisqu'il a fallu envoyer 21821 routes vers les deux pairs. Il a fallu 19,34 secondes pour envoyer ces messages vers le pair eBGP et 19,77 secondes vers le RR-client.



(a) Version Originale



(b) Première version modifiée



(c) Deuxième version modifiée

FIG. 4.22 – *Temps de traitement global II*

Première Version Lors du premier test, 4.21(b), *BGPd* a mis 4,37 secondes pour envoyer tous les messages UPDATE vers le pair eBGP et 4,18 secondes vers le RR-client.

Lors du deuxième test, 4.22(b), *BGPd* a mis 17,64 et 17,91 secondes pour annoncer toutes les routes vers, respectivement, le pair eBGP et le RR-client.

Deuxième Version La figure 4.21(c) montre les temps d'envoi des messages UPDATE. *BGPd* a mis 4,45 et 4,31 secondes pour envoyer tous les messages UPDATE vers, respectivement, le pair eBGP et le RR-client.

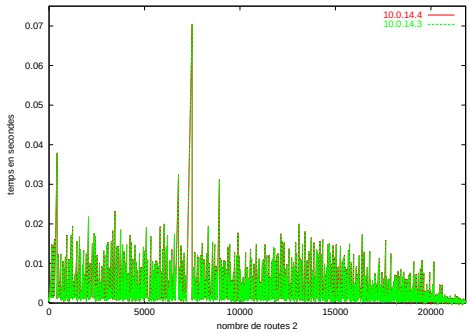
Le deuxième test, figure 4.22(c), prend naturellement plus de temps et il faut 21,70 et 21,52 secondes à *BGPd* pour envoyer tous les messages UPDATE vers, respectivement, le pair eBGP et le RR-client.

Comparaison On peut noter qu'il y a des délais introduits dans les graphes des versions modifiées. Il semble qu'à certains moments *BGPd* n'enverrait plus de manière régulière ses messages UPDATE. Cela peut se constater de par les figures 4.23 du test suivant où l'on ne voit plus apparaître ces délais.

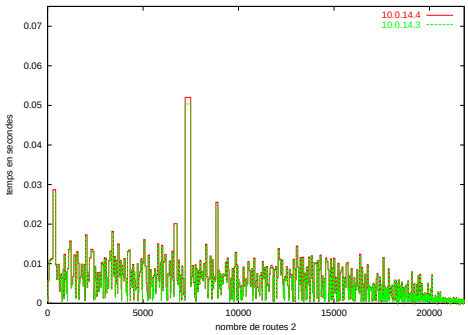
Quant au temps de traitement, ils sont plus grands pour la deuxième version de la modification jusqu'à dégradation des performances par rapport à la version originale dans le cas où le rapport entre le nombre d'attributs sur le nombre de préfixes est grand. Ce qui n'est pas le cas pour la première version de la modification. Quel que soit le rapport entre

les deux, cette version apporte une amélioration des performances avec, bien entendu, une plus grande amélioration quand le rapport est petit.

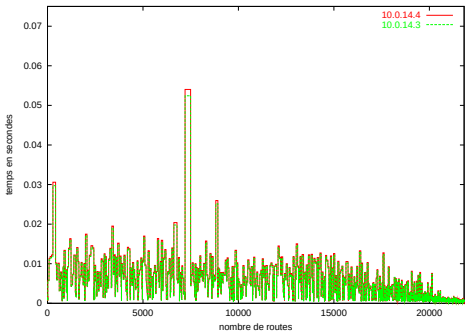
Résultats Les time-stamps restent les mêmes que ceux présentés à la page 46 pour la version originale et à la page 61 pour les deux versions modifiées.



(a) Version originale



(b) Première version modifiée



(c) Deuxième version modifiée

FIG. 4.23 – *Temps de traitement des routes*

Version Originale Le temps moyen de construction d'un message UPDATE, figure 4.23(a), pour le pair eBGP est de 0,0053 seconde et de 0,00546 pour le RR-client.

Lors du deuxième test, figure 4.24(a), le temps moyen de construction d'un message UPDATE est de 0,00072 seconde pour le pair eBGP et de 0,000825 pour le RR-client.

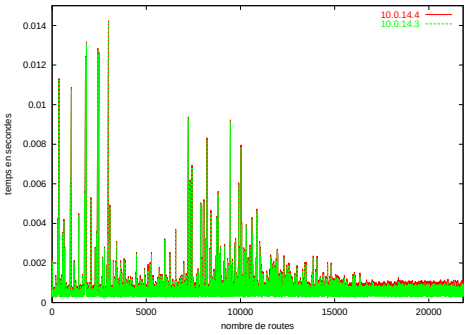
Première Version Lors du premier test, figure 4.23(b), le temps moyen mis par *BGPd* pour construire un message UPDATE est de 0,00573 et 0,00589 seconde pour, respectivement, le pair eBGP et le RR-client.

La figure 4.24(b) représente le deuxième test effectué pour lequel le temps moyen est de 0,00092 seconde pour le pair eBGP et de 0,00094 pour le RR-client.

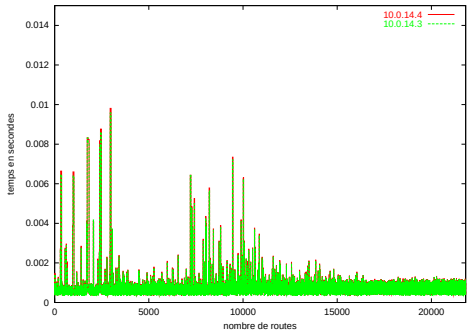
Deuxième Version Les temps moyens du premier test, figure 4.23(c), sont de 0,00757 et de 0,0078 seconde pour, respectivement, le pair eBGP et le RR-client.

Les temps moyens sont à nouveau supérieurs lors du deuxième test, figure 4.24(c). Ceux-ci sont de 0,00102 et de 0,00107 seconde pour le pair eBGP et le RR-client.

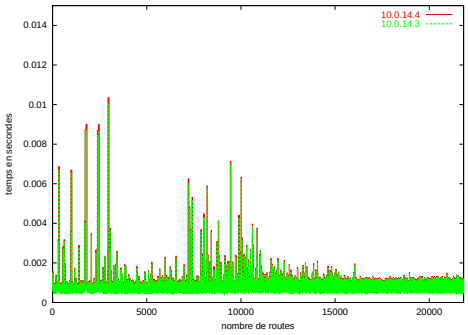
Comparaison On peut remarquer sur les figures 4.23 et 4.24 que le tracé ne ressemble plus du tout au tracé des tests précédents. Ceci est tout à fait normal puisque dans les tests précedents, *BGPd* recevait environ 1000 préfixes par message. Dans les tests effectués dans cette partie, il n'est plus question d'avoir des messages avec autant de préfixes. De plus, si il y a tant de variations dans les traitements, c'est tout simplement parce que le nombre de préfixes reçus dans les messages UPDATE varie beaucoup.



(a) Version originale



(b) Première version modifiée



(c) Deuxième version modifiée

FIG. 4.24 – *Temps de traitement des routes II*

4.6 Conclusion

Lors de ce chapitre, on a construit deux versions pour l'amélioration de *BGPd*. De par cette amélioration, *BGPd* a été capable d'envoyer des messages UPDATE contenant plusieurs préfixes. Ces deux versions ont été testées de la même manière que la version originale et ont pu montrer de grandes améliorations que ce soit pour la première version ou pour la seconde. Les deux versions améliorées prenaient environ deux fois moins de temps pour effectuer la même tâche que la version originale.

Cependant, les données injectées lors des tests n'étaient pas réelles puisque toutes les routes avaient les mêmes attributs. Ce genre de test avec des données non réelles nous a permis d'interpréter plus facilement le comportement de *BGPd*. On a tout de même

recommencé les mêmes tests pour les trois versions et ce, avec des données réelles. Ceux-ci ont permis de constater que le rapport du nombre d'attributs sur le nombre de préfixes influence fortement les temps de traitement de *BGPd*.

En conlusion, La deuxième version de la modification apporte des améliorations dans les conditions où le rapport entre le nombre d'attributs sur le nombre de préfixes est petit. Dans le cas d'une ouverture de session, il se peut que cette version puisse amener à des coupures de sessions entre des pairs BGP. Cette version ne peut donc pas être utilisée telle qu'implémentée actuellement. Il faudrait d'abord améliorer deux de ses caractéristiques :

1. Rendre plus efficace la fonction de création de la clé de hachage.
2. Eviter de gaspiller trop de mémoire vive en initialisant la taille des messages UPDATE en construction directement à leur taille maximale.

La première version de la modification, quant à elle, amène une amélioration des performances de manière globale. Néanmoins, dans le cas d'une ouverture de session et où le rapport entre le nombre d'attributs sur le nombre de préfixes est grand, cette modification peut dégrader légèrement les performances, en temps, de *BGPd*. Même si il y a une légère dégradation des performances au point de vue du temps, on a tout de même pu économiser un peu de bande passante en envoyant moins de messages UPDATE. De plus, ce point pourrait être amélioré en ajoutant le moyen de parcourir la table de routage BGP en fonction des attributs et non plus en fonction des préfixes.

Conclusion

Dans ce mémoire, on a étudié le protocole de routage inter-domaine BGP-4. Ce protocole est à l'heure actuelle le seul protocole permettant de faire fonctionner l'Internet en interconnectant tous les AS entre eux. Le fonctionnement et les performances de ce protocole sont donc en relation directe avec la qualité de service offert aux utilisateurs. Pourtant, il subsiste toujours des ombres au niveau du fonctionnement de BGP-4. Il s'avère que les nombreuses extensions apportées au protocole apportent aussi de nouveaux problèmes inconnus jusqu'alors mais d'autres conséquences graves peuvent apparaître lors de mauvaises configuration, de ruptures de lien ou lors de problèmes d'implémentation. On a donc été amené à s'intéresser à une implémentation open source de ce protocole faisant partie d'une suite de logiciels de routage *Zebra*.

Dans un premier temps on s'est intéressé au comportement et aux performances de ce logiciel. Pour ce faire, on a mis en place deux séries de tests permettant de comprendre le fonctionnement et de mesurer certains temps de traitement de ce logiciel. La première série de tests concernait l'ouverture de session entre deux pairs, ce qui peut s'apparenter à une ré-initialisation d'un pair après un crash ou à un rétablissement d'un lien. Cette série se proposait de mesurer le temps qu'il fallait à *Zebra* pour redistribuer sa LOC-RIB. Et la deuxième série de tests concernait le traitement des messages UPDATE jusqu'à la redistribution des routes reçues. Ces deux scénarios mis en place mesurent directement les temps de convergence des annonces avec le logiciel *Zebra*. L'établissement de ces tests a conduit, non seulement, à la mise au point de scénarios permettant la mise en oeuvre de ces tests mais aussi à l'élaboration d'un outil spécifiquement créé pour ce logiciel nous permettant d'effectuer les scénarios à mettre en place mais a été également élaboré de manière à pouvoir facilement ré-utiliser cet outil pour d'autres tests que ceux créés pour ce mémoire. Ces deux séries de tests accomplis, on a pu identifier un point faible de *Zebra*. En effet, ce logiciel ne construit jamais de messages UPDATE avec plusieurs préfixes dans le champ NLRI. Or ce fonctionnement a une conséquence néfaste directe sur l'utilisation du processeur par ce logiciel et sur la bande passante du réseau mais aussi pour les routeurs avoisinants.

L'amélioration apportée à *Zebra* a donc consisté à changer son comportement de telle façon que celui-ci essaye de mettre plusieurs préfixes dans un message UPDATE. Deux versions ont été implémentées. La première ne peut construire qu'un message UPDATE à la fois, ce qui implique qu'à chaque changement d'attributs entre deux routes consécutives passant par le Decision Process, le message en construction est envoyé. La deuxième version essaye quant à elle d'optimiser la taille des messages en lui permettant de pouvoir construire plusieurs messages UPDATE ayant des attributs différents.

Les premiers tests menés sur ces deux versions améliorées ont montré que celles-ci étaient bien plus performantes que la version originale. En général, les deux versions modifiées prenaient deux fois moins de temps que la version originale. La deuxième version permettant de construire plus d'un message UPDATE en même temps a toujours pris un peu plus de temps, bien qu'infime, que la première version car il a fallu un peu plus de code pour pouvoir gérer la construction simultanée de plusieurs messages UPDATE.

Une fois, les premiers test menés, on a décidé de confronter ces implémentations à des données réalistes. Les mêmes tests ont été effectués avec deux catégories de données différentes. La première catégorie de données possédait des routes ayant des attributs peu variés tandis que la deuxième catégorie était constituée de routes ayant des attributs très variés. Lors des tests avec les attributs peu différents, les conclusions sont à nouveau positives puisque les deux versions modifiées ont encore montré leur capacité à mieux gérer les situations que la version originale.

La deuxième série de tests se rapprochant du pire des cas, a sévèrement mis à l'épreuve les deux implémentations. En effet, la première version est toujours plus performante que la version originale quant il s'agit de traiter des messages UPDATE mais, par contre, lors du test d'ouverture de session avec un pair, on a pu constater que les performances au niveau du temps s'étaient un peu dégradées. Cependant, deux points sont à souligner. Premièrement, même si les performances au point de vue temps se sont un peu dégradés, il n'en reste pas moins que cette version améliore l'utilisation de la bande passante, puisqu'elle envoie de toute façon moins de messages UPDATE que la version originale. Deuxièmement, des modifications supplémentaires au logiciel permettrait peut-être d'absorber cette dégradation en dotant *Zebra* de la capacité de parcourir sa table de routage BGP en fonction des attributs et non plus en fonction des préfixes. Cette modification n'a pas pu être apportée pour des raisons de temps.

La deuxième version, quant à elle, obtient des performances moindres que lors des tests précédents. Tout comme pour la première version, il n'en reste pas moins que l'utilisation de la bande passante n'est pas gaspillée comme avec la version originale. Lors d'une ouverture de session, cette version amène jusqu'à une coupure de session car le temps de traitement est trop long. Cependant, deux points amenant à cette situation ont été identifiés et pourraient être améliorés par la suite. Premièrement, la fonction de création de la clé de hachage n'est pas efficace. Et, deuxièmement, il y a un gaspillage de mémoire vive en initialisant la taille d'un message UPDATE en construction à la taille maximale prévue par le protocole. L'amélioration concernant ces deux points n'a pas non plus été entamée, comme pour la première version, pour des raisons de temps.

Bibliographie

Références

[Bat96] E. Chen, T. Bates. An application of the bgp community attribute in multi-home routing. RFC1998, Aout 1996.

[Bon02] B. Quoitin, O. Bonaventure. A survey of the utilization of the bgp community attribute. draft-quoitin-bgp-comm-survey-00, 2002. work in progress.

[Bou01] N. Boulay. Hack en c. *Linux Magazine*, (32) :29 – 41, Octobre 2001.

[Bra89] H-W. Braun. Models of policy based routing. RFC1104, 1989.

[Che00a] E. Chen. Route refresh capability for bgp-4. RFC2918, Septembre 2000.

[Che00b] T. Bates, R. Chandra, E. Chen. BGP route reflection - an alternative to full mesh ibgp. RFC 2796, Avril 2000.

[Che02a] Q. Vohra, E. Chen. BGP support for four-octet as number space. draft-ietf-idr-as4bytes-05, mai 2002. work in progress.

[Che02b] S. R. Sangli, Y. Rekhter, R. Fernando, J. G. Scudder, E. Chen. Graceful restart mechanism for bgp-4. draft-ietf-idr-restart-05, 2002. work in progress.

[Dee98] A. Conta, S. Deering. Internet control message protocol (icmpv6) specification. RFC2463, Décembre 1998.

[Dup99] P. Marques, F. Dupont. Use of bgp-4 multiprotocol extensions for ipv6 inter-domain routing. RFC2545, Mars 1999.

[ea02] E. Rosen, et al. BGP/mpls vpns. draft-ietf-ppvpn-rfc2547bis-01, janvier 2002. work in progress.

[ESN01a] ESNet. BGP *Route Oscillations Type 1a.* http ://www.es.net/hypertext/-welcome/pr/BGP/BGP_Route_Oscillation_Type1a_v1.0.pdf, 2001.

[ESN01b] ESNet. BGP *Route Oscillations Type 1b.* http ://www.es.net/hypertext/-welcome/pr/BGP/BGP_Route_Oscillation_Type1b_v1.1.pdf, 2001.

[ESN01c] ESNet. *Stable Routing Loops.* http ://www.es.net/hypertext/welcome/pr/BGP/-BGP_Routing_Loop_v1.1.pdf, 2001.

[Gov98] C. Villamizar, R. Chandra, R. Govindan. BGP route flap damping. RFC2439, Novembre 1998.

[Has95] D. Haskin. A bgp/idrp route server alternative to a full mesh routing. RFC1863, 1995.

[Hef02] A. Hefferman. Protection of bgp sessions via the tcp md5 signature option. draft-ietf-idr-rfc2385bis-01, Mars 2002.

[Ibe97] O. C. Ibe. *Essentials of ATM Networks and Services.* Addison Wesley, 1997.

[III99] J. W. Stewart III. *BGP4 Inter-Domain Routing in the Internet.* The Addison-Wesley Networking Basics Series, 1999.

[Li95] Y. Rekhter, T. Li. A border gateway protocol 4 (bgp-4). RFC1771, Mai 1995.

[Li96] R. Chandra, P. Traina, T. Li. BGP communities attribute. RFC1997, Aout 1996.

[Mae02] O. Maennel. *Generating realistic routing tables in a lab.* http ://www.olafm.de/-2001/networking/index.html, 2002.

[oMN01] University of Michigan and Merit Network. *MRT user guide.* http ://www.merit.edu/ mrt/mrt_doc/mrtuser.pdf, 2001.

[P.T95] P.Traina. BGP-4 protocol analysis. RFC1774, Mars 1995.

[Puj98] G. Pujolle. *Les réseaux.* Eyrolles, 2ème edition, 1998.

[Ram01] E. Chen, S. Ramachandra. Address prefix based outbound route filter for bgp-4. draft-chen-bgp-prefix-orf-03, Octobre 2001. work in progress.

[Rek93] H-W. Braun, P. S. Ford, Y. Rekhter. *CIDR and evolution of IP.* http ://www.caida.org/outreach/papers/1993/cei/inet93.cidr.pdf, 1993.

[Rek02a] E. Chen, Y. Rekhter. Cooperative route filtering capability for bgp-4. draf-ietf-idr-route-filter-05, janvier 2002. work in progress.

[Rek02b] S. R. Sangli, D. Tappan, Y. Rekhter. BGP extended communities attribute. draft-ietf-idr-bgp-ext-communities-05, 2002. work in progress.

[Rek02c] T. Bates, R. Chandra, D. Katz, Y. Rekhter. Multiprotocol extensions for bgp-4. draft-ietf-idr-rfc2858bis-02, avril 2002.

[Rek02d] Y. Rekhter. A border gateway protocol. draft-ietf-idr-bgp4-17.txt, 2002.

[Ret02] D. McPherson, V. Gill, D. Walton, A. Retana. BGP persistent route oscillation condition. draft-ietf-idr-route-oscillation-01, 2002. work in progress.

[Rit88] B. W. Kernighan , D. M. Ritchie. *Le langage C Norme ANSI.* Masson - Prentice-Hall, 2eme edition, 1988.

[Ros01] Y. Rekhter, E. Rosen. Carrying label information in bgp-4. RFC3107, Mai 2001.

[San02] E. Chen, S. Sangli. Dynamic capability for bgp-4. draft-ietf-idr-dynamic-cap-01, janvier 2002. work in progress.

[Scu01] P. Traina, D. McPherson, J. Scudder. Autonomous system confederations for bgp. RFC3065, Février 2001.

[Scu02] R. Chandra, J. Scudder. Capabilities advertisement with bgp-4. draft-ietf-idr-rfc2842bis-02, avril 2002.

[Ste94] W. R. Stevens. *TCP/IP illustrated*, volume 1. Addison-Wesley, 1994.

[Ste99] E. Chen, J. Stewart. A framework for inter-domain route aggregation. RFC2519, Février 1999.

[Tan96] A. Tanenbaum. *Réseaux.* Prentice-Hall Dunod, 3ème edition, 1996.

[Tou99] L. Toutain. *Réseaux locaux et internet des protocoles à l'interconnexion.* Hermes, 2ème edition, 1999.

[Tra95] P. Traina. Experience with the bgp-4 protocol. RFC1773, Mars 1995.

[Var93a] V. Fuller, T. Li, J. Yu, K. Varadhan. Cidr : an address assignment and aggregation strategy. RFC1519, 1993.

[Var93b] K. Vardhan. BGP OSPF interaction. RFC1403, janvier 1993.

[War02] G. Nalawade, J. Scudder, D. Ward. BGP inform message. draft-nalawade-bgp-inform-00, 2002. work in progress.

[Wet02] T. Anderson, S. Shenker, I. Stoica, D. Wetherall. *Towards More Robust Internet Protocols.* http ://www.cs.washington.edu/masters/hype/att-0030/01-robust.ps, 2002.

Annexe A

Structures principales de BGPd

Il existe deux structures sur lesquelles repose BGPd, une structure permettant de retenir les informations nécessaires sur les pairs avec lesquels BGPd communique et une autre permettant de représenter une RIB. Bien sûr, il n'existe pas que ces structures là dans l'implémentation. Pourtant, il ne sera pas fait mention des structures dont on ne parle pas dans la suite de ce travail.

A.1 Représentation d'une RIB

La structure la plus importante de BGPd est celle qui permet de représenter une RIB. Cette RIB, dans *Zebra*, est représentée par un arbre équilibré (cfr. Figure A.1). Celle-ci est constituée de noeuds et chaque noeud contient les pointeurs nécessaires pour représenter l'arbre :

- un pointeur vers la table de routage,
- un pointeur vers son père,
- deux autres pointeurs, chacun pointant vers un de ses fils.

En plus de ces champs, chaque noeud possède :

- un champ pour le préfixe,
- un champ servant de mutex,
- et un pointeur vers une liste d'informations en rapport avec ce préfixe.

Cette liste, appelée bgp_info, est doublement chaînée, et reprend les informations en fonction d'un pair et dans chacune de ces structures il y a :

- la métrique IGP,
- une structure permettant de collecter les infos pour le route flap dampening,
- le temps depuis lequel cette route est valide,
- le type et sous-type,
- le champ suppress permet de savoir si la route a été supprimée à cause d'une agrégation,
- le champ flag : c'est par ce champ que l'on peut déterminer si la route est sélectionnée,
- un pointeur vers une structure contenant les attributs de ce préfixe.

La structure attr contient non seulement les informations requises suivantes :

- l'origine pouvant être IGP, EGP ou INCOMPLETE,
- le prochain saut,
- le Multi Exit Disc,
- la préférence locale,
- le numéro d'AS qui a agrégé le préfixe,
- l'adresse de l'AS qui a agrégé le préfixe,
- le poids associé à la route,
- l'id de l'origine de l'annonce pour les RR,
- une liste de cluster en cas de RR,
- un pointeur vers l'AS path,
- un pointeur vers une communauté,
- un pointeur vers une communauté étendue.

Mais également :

- un compteur de référence permettant de ne pas dupliquer cette structure quand deux préfixes ont les mêmes attributs,

- un champ flag permettant de déterminer quels sont les attributs dont les valeurs sont valides dans cette structure,
- un pointeur vers une structure pour les attributs non reconnus mais transitifs,
- trois champs utilisés pour le MPLS.

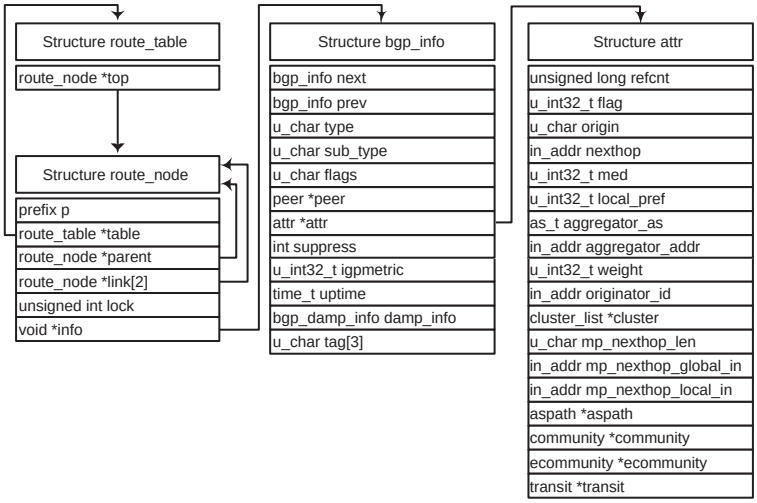


FIG. A.1 – *Structure d'une RIB*

A.2 La structure peer

Les RIB-IN et RIB-OUT se retrouvent dans chaque structure permettant de retenir les informations utiles à propos des pairs avec lesquels BGPd communique. Cette structure, appelée peer (cfr. Figure A.2), nous permet d'accéder aux informations suivantes :

- les valeurs des timers : start, connect, holdtime, keepalive, ...
- des champs pour le statut,
- des champs statistiques sur notamment le nombre de connexions établies et droppées,
- des informations pour la connexion TCP.

Mais surtout ces informations-ci aussi :

- le numéro de l'AS distant,
- le numéro de l'AS local,
- le router-id distant,
- le router-id local,
- un pointeur vers une structure stream permettant d'accueillir un paquet lu,
- un pointeur vers un structure stream_fifo permettant de mettre les messages à envoyer vers ce pair,
- L'ADJ-RIB-IN et l'ADJ-RIB-OUT.

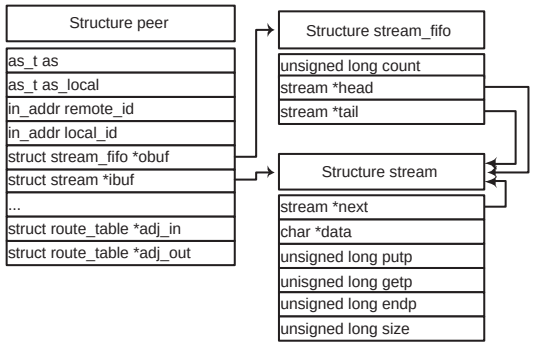


FIG. A.2 – *Structure peer et stream, stream_fifo*

Description des messages de BGP4

Le protocole de base BGP définit 4 messages dont une description est donnée dans la suite de cette section. Cependant, d'autres messages ont été définis par des extensions du protocole. Le format de ces messages sont tous décrits dans cette annexe, ainsi qu'une description de leur sémantique.

B.1 HEADER

Chaque message décrit par la suite doit commencer par l'en-tête suivant :

Nom du champ	longueur
Marker	16 octets
Length	2 octets
Type	1 octet

FIG. B.1 – *Format de l'en-tête*

- *Marker* : contient une valeur que celui qui le réceptionne sait prédire. Sert pour l'authentification et pour la détection de désynchronisation entre deux pairs.
- *Length* : la valeur de ce champ représente la longueur totale du message (header inclus). Le paquet doit obligatoirement avoir une longueur au moins égale à 19 et au plus à 4096.
- *Type* : la valeur de ce champ définit le type du message.
  1. OPEN
  2. UPDATE
  3. NOTIFICATION
  4. KEEPALIVE
  5. ROUTE REFRESH
  6. DYNAMIC CAPABILITY
  7. (TBD) INFORM

B.2 OPEN message

Ce message est le premier message envoyé à un autre pair BGP lorsque l'on veut initier une session avec lui. C'est notammment lors de l'envoi de ce message que les deux pairs BGP vont pouvoir négocier certaines options offertes par BGP ou extensions de BGP [Scu02].

Lorsqu'un pair reçoit un message OPEN valide et qu'il est en droit de le recevoir au vu de l'état courant de la machine à état, alors il doit renvoyer un message KEEPALIVE en confirmation de la réception de ce message.

- *Version* : valeur indiquant le numéro de version du protocole BGP. Le numéro de version actuel est 4.
- *My Autonomous System* : valeur indiquant le numéro de système autonome (AS) de l'émetteur du message.
- *Hold Time* : proposition de valeur pour le temps maximal pouvant s'écouler sans recevoir de message KEEPALIVE ou UPDATE
- *BGP Identifier* : valeur utilisée pour identifier le pair BGP.
- *Option Parameter Length* : longueur du champ *Option Parameter*.

Nom du champ	longueur
Version	1 octet
My Autonomous System	2 octets
Hold Time	2 octets
BGP Identifier	4 octets
Option Parameter Length	1 octet
Option Parameter	variable

FIG. B.2 – *Format du message OPEN*

– *Option Parameter* : Chaque paramètre est encodé de la façon suivante :

Nom du champ	longueur
Parameter Type	1 octet
Parameter Length	1 octet
Parameter Value	variable

FIG. B.3 – *Format des options*

Dans le draft décrivant le protocole BGP, il n'y a qu'une option définie. Cette option a 1 comme valeur pour le *Parameter Type* et est utilisé pour le mécanisme d'authentification. Le *Parameter Length* contient le code d'authentification et le *Parameter Value* contient les données d'authentification.
Il existe un défaut majeur à ce mécanisme de négociation de capacités ; il faut absolument couper la session entre les deux pairs si l'on veut rajouter une capacité ou en enlever une. Donc, pour améliorer cette fonctionnalité, un message a été rajouté. Celui-ci permet de négocier dynamiquement les capacités de BGP-4 [San02]. La création de ce message n'a toutefois pas enlevé à BGP-4 la capacité de négocier certaines options avec le message OPEN.

B.3 UPDATE message

Il s'agit du message de BGP par lequel on va pouvoir annoncer plusieurs routes que l'on peut joindre et partageant les mêmes attributs et/ou annuler certaines de ces routes (ce qui revient à dire qu'elles ne sont plus accessibles à partir du pair qui les avaient annoncer). Si on doit annoncer plusieurs routes joignables mais qu'elles n'ont pas les mêmes attributs, il faut alors les annoncer par plusieurs de ces messages UPDATE.

format du message UPDATE Il ne faut pas oublier qu'il y a toujours le préambule à ajouter au format décrit ci-dessous (voir figure B.1).

– *Withdrawn Routes Length* : il s'agit de la longueur du champ contenant les routes à annuler.
– *Withdrawn Routes* : il s'agit des routes à annuler selon le format suivant :

  – *length* : longueur du préfixe en nombre de bits.
  – *prefix* : valeur d'une adresse IP. Cette valeur doit être finie par des zéros pour obtenir un alignement sur un octet.

  Ce champ a une longueur égale à la valeur du champ *Withdrawn Routes Length*.

Nom du champ	longueur
Withdrawn Routes Length	2 octets
Withdrawn Routes	variable
Total Path Attribute Length	2 octets
Path Attributes	variable
Network Layer Reachability Information (NLRI)	variable

FIG. B.4 – *Format du message UPDATE*

Nom du champ	longueur
Length	1 octet
Prefix	variable

FIG. B.5 – *Format d'un préfixe*

- *Total Path Attributes Length* : il s'agit de la longueur du champ *Path Attributes*.
- *Path Attributes* : Ce champ définit tous les attributs concernant les préfixes annoncés dans le *NLRI*. Ces attributs sont les éléments importants, avec le préfixe, du fonctionnement de BGP. C'est entre autre grâce à une partie de ceux-ci que l'on va pouvoir déterminer quel va être la meilleure route.
  Tous les attributs ont le format de la table B.6 :

Nom du champ	longueur
Attribute Type	2 octets
Attribute Length	1 or 2 octets
Attribute Value	variable

FIG. B.6 – *Format des attributs*

- *Attribute Type* : Il s'agit d'un champ sur deux octets. Le premier octet définit l'*Attribute Flags* et le second définit l'*Attribute Type Code*. Comme le montre la figure B.7, les 4 bits de poids fort de l'*Attribute Flags* ont une signification tandis que les autres doivent obligatoirement être mis à 0 car ils sont inutilisés.

Optional	Transitive	Partial	Extended Length	0000

FIG. B.7 – *Format de l'Attribute Flags*

  - *Optional* bit : Ce bit détermine si l'attribut est "bien connu" (valeur à 0) ou optionnel (valeur à 1). Si l'attribut est "bien connu" alors toutes les implémentations doivent savoir le traiter. Tandis que si il est optionnel, il n'est pas obligatoire qu'une implémentation le reconnaisse.
  - *Transitive* bit : Pour les attributs "bien connu", ce bit doit être mis à 1. Cela signifie que lorsqu'une route contient un attribut transitif, celui-ci fait obligatoirement partie de cette route quand celle-ci est redistribuée à d'autres paris.
  - *Partial* bit : Ce bit est mis à 1 lorsqu'un routeur BGP reçoit un préfixe avec un attribut optionnel transitif et qu'il ne sait pas traiter l'attribut. Comme cet attribut est transitif, il doit faire suivre le préfixe vers un autre pair avec cet attribut dans le *Path Attribute*. Ainsi, l'autre pair peut déterminer qu'il y a eu une perte d'information dans cette annonce. Il est mis obligatoirement à 0 si

l'attribut est "bien connu" ou optionnel et non transitif.
  - *Extended Length* bit : Si le bit est à 0 alors l'*Attribute Length* est d'une longueur d'un octet. Si ce bit est à 1 alors l'*Attribute Length* est de deux octets.

  L'*Attribute Type Code* est un nombre assigné par l'IANA qui identifie l'attribut.
  - *Attribute Length* : la valeur de ce champ représente la longueur de l'*Attribute Value*.
  - *Attribute Value* : représente la valeur de l'attribut en fonction du code de celui-ci.
- NLRI : Ce champ est prévu pour contenir une liste de préfixes d'adresses IP et est encodé de la manière suivante :

Nom du champ	longueur
Length	1 octet
Prefix	variable

FIG. B.8 – *Format du NLRI*

Le champ *Length* détermine la longueur du champ *Prefix*. La valeur de ce champ doit être comprise entre 0 et 4. Le champ *Prefix* contient un préfixe d'une adresse IP.
La longueur totale du NLRI peut être calculée selon la formule suivante :
UPDATE Message Length−23− *Total Path Attributes Length*− *Withdrawn Routes Length*

Le message UPDATE peut annoncer plusieurs préfixes dans le même message seulement si ceux-ci ont exactement les mêmes attributs. Ce qui revient à dire que deux préfixes à annoncer n'ayant pas les mêmes attributs sont annoncés dans des messages UPDATE distincts. De même, un message UPDATE peut contenir plusieurs préfixes à annuler.

B.4 NOTIFICATION message

Le message NOTIFICATION indique une erreur de déroulement dans le protocole défini par BGP-4. Lors de l'envoi d'un tel message, BGP rompt la session avec le pair pour lequel il envoie ce message. Il en va donc de même lors de la réception d'un tel message.

Nom du champ	longueur
Error Code	1 octet
Error subcode	1 octet
Data	variable

FIG. B.9 – *Format message NOTIFICATION*

Le message NOTIFICATION est composé de 3 champs :

1. *Error Code* : Ce champ structure les erreurs en grosse catégorie. Celle-ci sont au nombre de 6 :
   (a) Message Header error
   (b) OPEN message error
   (c) UPDATE message error
   (d) Hold Timer Expired
   (e) Finite State Machine Error
   (f) Cease
2. *Error subcode* : Le sous-code indique précise quel est le genre de l'erreur qui est survenue. Pour chaque cétagorie citée ci-dessus, il existe des sous-codes sépcifiques :

- sous-code pour l'HEADER
  (a) Connection not synchronized
  (b) Bad Message Length
  (c) Bad Message Type
- sous-code du message OPEN
  (a) Unsupported Version Number
  (b) Bad Peer AS
  (c) Bad BGP Identifier
  (d) Unsupported Optional Parameter
  (e) Authentication Failure
  (f) Unacceptable Hold Time
- sous-codes pour le message UPDATE
  (a) Malformed Attribute List
  (b) Unrecognized Well-known Attribute
  (c) Missing Well-known Attribute
  (d) Attribute Flags Error
  (e) Attribute Length Error
  (f) Invalid ORIGIN Attribute
  (g) Invalid NEXT-HOP Attribute
  (h) Optional Attribute Error
  (i) Invalid Network Error
  (j) Malformed AS-PATH

3. *Data* : Le champ *Data* est un champ de longueur variable pouvant contenir un partie du message qui est à l'origine de l'envoi du message NOTIFICATION.

B.5 KEEPALIVE message

Le procole BGP-4 n'utilise pas un moyen de détecter les coupures de liaison basé sur la couche transport. De ce fait, BGP-4 utilise un message qui est envoyé vers un pair BGP à intervalles réguliers si aucun message (à part l'OPEN) de sa part n'est réceptionné et dont la seule utilité est d'indiquer sa présence à un autre pair ou d'effectivement remarquer qu'il n'existe plus de liaison entre les deux pairs. Ce message s'appelle le message KEEPALIVE et est juste composé de l'HEADER décrit précédemment (voir B.1).

Si la valeur du champ *Hold Time* présent dans le message OPEN lors de l'ouverture de la session entre deux pairs est à 0, alors il ne faut pas envoyer de message KEEPALIVE.

B.6 INFORM message

Le message INFORM est utilisé pour prévenir un pair qu'une erreur s'est produite et que celle-ci est assez mineure pour que la session ne soit pas coupée.

Nom du champ	longueur
Event Code	2 octets
Type	1 octet
Length	2 octets
Value	variable

FIG. B.10 – *Format du message INFORM*

- *Event Code* : Ce champ définit un type d'événement pour lequel le message INFORM a été envoyé.
- *Type, Length, Value* : de l'information est apportée avec le message INFORM, et c'est par ce triplet qu'un pair peut déterminer quel est le type de l'information qui est transportée et la longueur de cette information.

B.7 ROUTE REFRESH message

Le message ROUTE REFRESH est utilisé pour indiquer à un pair qu'il souhaite qu'un pair redistribue toutes ses routes à l'émetteur de ce message. Cette capacité doit être négociée pour pouvoir l'utiliser [Scu02].

Nom du champ	longueur
AFI	2 octets
Res.	1 octet
SAFI	1 octets

FIG. B.11 – *Format du message ROUTE REFRESH*

- AFI : indique quel est l'identifiant de la famille d'adresse utilisée.
- Res. : cet octet est réservé.
- SAFI : indique quel est le subsequent address family identifier[43].

B.8 DYNAMIC CAPABILITY message

Le message de capacité dynamique est utilisé pour subvenir à une faiblesse de BGP-4. En effet, tout au long du développement du protocole BGP-4 de nouvelles capacités sont apparues. Ces capacités, pour ne pas rendre obsolètes les implémentations déjà existantes, ont été considérées comme optionnelles. Le seul moyen de pouvoir en disposer était de négocier leur utilisation lors d'une ouverture de session. L'inconvénient majeur de cette situation est que l'on ne pouvait pas négocier un ajout/retrait d'une capacité lors d'une session déjà établie entre deux pairs. Avec ce nouveau message, les routeurs peuvent désormais négocier l'ajout/retrait de capacités lorsqu'une session est déjà établie entre deux routeurs BGP-4.

[43] Voir le draft définissant le multiprotocol extension pour les définitions des valeurs permises [Rek02c].

Nom du champ	longueur
Action	1 octet
Capability Code	1 octet
Capability Length	1 octet
Capability Value	variable

FIG. B.12 – *Format du message de négociation dynamique de capacité*

- *Action* : Ce champ a pour valeur 0 pour annoncer une capacité, 1 pour annuler une capacité.
- Le triplet <*Capability Code, Capability Length, Capability Value*> : Ces trois champs doivent comporter les valeurs définies par le RFC2842 [Scu02].

Annexe C : Code source

C.1 Code de l'outil d'évaluation de performance

C.1.1 PerfEval.h

```
/* Header file for the structures needed for the evaluation of performance */
#ifndef PerfEval_H
#define PerfEval_H

#ifdef MODIF_SEB

#include <sys/time.h>
#include <zebra.h>
#include "eval_sema.h"
#include "bgpd/bgpd.h"
#include "bgpd/bgp_community.h"
#include "bgpd/bgp_attr.h"
#include "lib/linklist.h"
#include "lib/stream.h"
#include "lib/prefix.h"


#define MaxElt 500

unsigned char StartEvaluation2;

#define OPEN_MSG      0
#define UPDATE_MSG    1
#define WITHDRAW_MSG  2
#define NOTIFY_MSG    3

//value of the community for the begin of the test
```

```
#define COMMUNITY_S 0X00ED0011 //237:17
//value of the community for the end of the test
#define COMMUNITY_W 0X00EE0012 //238:18
//number of peers for which we send packets to during the test
#define SEMA_N 2

/* values to create the semaphore
  the file is the default config file for bgpd */
#define FI_SEMA "/usr/local/etc/bgpd.conf"
#define ANY_SEMA 't'


/*******************************************************/
//structures containing data about the timestamps
/*******************************************************/

/*top of the list of TimePerf */
struct TopTimePerf
{
  struct timePerf *TP;
  int Count;
};

/* structure for time-stamp and describing this point
used to calculate how many time last the execution of
a part of code*/
struct timePerf
{
    unsigned long long int Time1;
    unsigned long long int Time2;
    char *Descr;
```

```
    struct timePerf *next;
};

/*******************************************************/
//structure containing data about the routes and packets
/*******************************************************/

/* structure containing prefix
used as a list with its 'next' field*/
struct prefix_eval
{
  //length of the prefix
  int len;
  //the prefix itself (char formatted)
  char val[20];
  //time spent by the code related to a prefix
  struct TopTimePerf TTP;
  struct prefix_eval *next;
};

/* top of the list of prefixes
*/
struct list_prefix_top
{
  //number of prefixes in the structure
  int  nbr;
  //pointing to the begin of the list
  struct prefix_eval *Prf;
};

/*structure containing packet*/
```

```
struct packetUpd {
    //contains the packet
    char *data;
    //size of the packet
    unsigned long size;
};

/* structure for performance evaluation, containing a packet */
struct perf {
    //type of the packet
    unsigned char type;
    //time-stamp associated with the packet
    unsigned long long int time1;
    //prefixes contained in the packet
    struct list_prefix_top PrfTop;
    //time spent by the code associated with the packets
    struct TopTimePerf TTP;
    //contains the packet
    struct packetUpd pack;
};

/*structure representing a perf list (the name has to be changed with struct listPerf)*/
struct Topperf
{
    //array containing packets and their prefixes
    struct perf *Tab[MaxElt];
    //the next Topperf
    struct Topperf *next;
    //the previous Topperf
    struct Topperf *prev;
};
```

```
/*****************************************************/
//Top of the structure containing all the data
/*****************************************************/
/* "list" structure (the name has to be changed with struct Topperf) */
struct listPerf {
        //File pointer associated with the structure where to save the datas
    FILE *PtrFi;
        //used as an event flag (to know when saving infos when sending datas)
        //may be used as an other event
    unsigned char WriteOK;
        //used as an event flag (to know when saving infos when putting info in the sending queue)
        //may be used as an other event
    unsigned char SendOK;
        //id of the semaphore(s)
    int idsem;
        //address of the host
    struct in_addr adr;
        //list of table containing all the data about the packets and routes
    struct Topperf *HeadPerf;
        //number of element present in the table
    unsigned int Count;
};

inline unsigned long long int GetTick ();
struct prefix_eval *last_prefix(struct list_prefix_top *PrfTop);
struct prefix_eval *InitAddRoute_Mult(struct perf *perf);
struct prefix_eval *InitAddRoute(struct peer *peer);
void AddRoute(struct prefix_eval *PtrPrf, char *Prefix, int len);
struct prefix_eval *AddRoute_nlri(struct prefix_eval *PtrPrf, char *Prefix, int len, int CountPrf);
```

```
void UpdateRoute_nlri(struct peer *peer, struct prefix_eval *PtrPrf);
void UpdateRoute_nlri_Mult(struct perf *perf, struct prefix_eval *PtrPrf);
int writeok(struct stream *s, char *Pref, int Len, int MaskP);
struct Topperf *SeekFirstTimeNonNull(struct listPerf UpdateRcvd, int *CountTmp);
int CmpCommunity(struct stream *s, u_int32_t community_val);
int comp_community(struct stream *s, bgp_size_t attr_len, struct community *community);
void DecrementeSema_and_FlushPeer(struct peer *peer);
// function saving structures in a file
void flush_all_peer(struct peer *Peer);
void flush_peer(struct peer *Peer);
void free_struct(struct perf *Perf);
void init_listPerf(char *buf, struct in_addr id, struct listPerf *UpdateRcvd);
struct perf *perf_init();
void put_packet(struct perf *perf, struct stream *packet, int size);
struct perf *copypacket(struct stream *packet, int size);
void SetTimePacket(struct perf *DataUpdate, long long int Time);
void SetTypePacket(struct perf *DataUpdate, unsigned char Type);
void SetDataUpdate(struct peer *peer, struct perf *DataUpdate);
void AddOneCountPrfTop(struct peer *peer);
void SetCountPrefTop(struct perf *perf, int Cnt);
void SetCountPrfTop(struct peer *peer, int Cnt);
void AddTime1(struct TopTimePerf *TPPerf, unsigned long long Time);
void AddTime2(struct TopTimePerf *TPPerf, unsigned long long Time);
void AddTopperf(struct listPerf *UpdateRcvd);
extern int community_compare (const void *a1, const void *a2);

#endif //MODIF_SEB
#endif //PerfEval_H
```

C.1.2 PerfEval.c

```
#include <zebra.h>

#include "bgpd/PerfEval.h"
#ifdef MODIF_SEB

#include "bgpd/bgpd.h"
#include "bgpd/bgp_community.h"
#include "bgpd/bgp_attr.h"
#include "lib/linklist.h"
#include "lib/stream.h"
#include "lib/prefix.h"
#include "eval_sema.h"

/************************************************************/
/************************ PREFIXES **************************/
/************************************************************/


//initialization to do before the test (in fact when a peer is created)
void init_listPerf(char *buf, struct in_addr id, struct listPerf *UpdateRcvd)
{
  char Fichier[256];

  if ((UpdateRcvd->idsem = CreationSema(CreationCleSema(FI_SEMA, ANY_SEMA), 1)) == -1)
       printf("CreationCleSema error\n");

  if (SetValeurSema(UpdateRcvd->idsem, 0, SEMA_N) == -1)
       printf("SetValeur error\n");
```

```
  UpdateRcvd->adr = id;
  UpdateRcvd->WriteOK = 0;
  UpdateRcvd->SendOK = 0;

  //name of the file where the info is written
  sprintf(Fichier, "/usr/local/etc/%s.dat", buf);

  UpdateRcvd->PtrFi = fopen(Fichier, "w+");
  UpdateRcvd->HeadPerf = NULL;
  AddTopperf(UpdateRcvd);
}

//return the pointer of the last element of the link list of prefix_eval
struct prefix_eval *last_prefix(struct list_prefix_top *PrfTop)
{
  struct prefix_eval *PtrPrf = PrfTop->Prf;
  struct prefix_eval *PtrPrfPrev = PtrPrf;
  while (PtrPrf != NULL)
  {
    PtrPrfPrev = PtrPrf;
    PtrPrf = PtrPrf->next;
  }
  return PtrPrfPrev;
}

//init of a structure prefix_eval (used before recording a prefix)
struct prefix_eval *InitAddRoute_Mult(struct perf *perf)
{
  struct prefix_eval *p_e;
```

```
  SetCountPrefTop(perf, 0);
  p_e = (struct prefix_eval *)malloc(sizeof(struct prefix_eval));
  p_e->next = NULL;
  p_e->TTP.Count = 0;
  p_e->TTP.TP = NULL;
  return p_e;
}

//Init of structure prefix_eval (to use before recording a route)
//obsoleted by InitAddRoute_Mult
//created when BGPd did not send several prefixes in the same message
struct prefix_eval *InitAddRoute(struct peer *peer)
{
       struct prefix_eval *p_e;
       //CountPrf = 0;
       SetCountPrfTop(peer, 0);
       p_e = (struct prefix_eval *)malloc(sizeof(struct prefix_eval));
       p_e->next = NULL;
       p_e->TTP.Count = 0;
       p_e->TTP.TP = NULL;
       return p_e;
}

//add a prefix to the structure prefix_eval
void AddRoute(struct prefix_eval *PtrPrf, char *Prefix, int len)
{
       PtrPrf->len = len;
       strcpy(PtrPrf->val, Prefix);
}

//add a prefix to a prefix_eval structure and allocate memory if needed
```

```
//obsoleted AddRoute
//created when BGPd did not send several prefixes in the same message
struct prefix_eval *AddRoute_nlri(struct prefix_eval *PtrPrf, char *Prefix, int len, int CountPrf)
{
       struct prefix_eval *PrfNext;
       if (CountPrf)
       {
              PrfNext = (struct prefix_eval *)malloc(sizeof(struct prefix_eval));
              PrfNext->TTP.Count = 0;
              PrfNext->TTP.TP = NULL;
              PrfNext->next = NULL;
              PtrPrf->next = PrfNext;
              PtrPrf = PrfNext;
       }
       AddRoute(PtrPrf, Prefix, len);

       return PtrPrf;
}

//attach the prefix_eval to list_prefix_top
void UpdateRoute_nlri_Mult(struct perf *perf, struct prefix_eval *PtrPrf)
{
  perf->PrfTop.Prf = PtrPrf;
}

//Update the information of the bgp peer after parsing the routes
//obsoleted by UpdateRoute_nlri_Mult
//created when BGPd did not send several prefixes in the same message
void UpdateRoute_nlri(struct peer *peer, struct prefix_eval *PtrPrf)
{
       int Cpt = peer->UpdateRcvd.Count-1;
```

```
        struct Topperf *PtrTopperf = peer->UpdateRcvd.HeadPerf;

        //if there are routes saved, attach it to perf structure
        //else there are no routes saved so delete the structure
        if (PtrTopperf->Tab[Cpt]->PrfTop.nbr > 0)
           PtrTopperf->Tab[Cpt]->PrfTop.Prf = PtrPrf;
        else
        {
           free(PtrTopperf->Tab[Cpt]);
           peer->UpdateRcvd.Count--;
           free(PtrPrf);
        }
}

//init a new structure perf
struct perf *perf_init()
{
  struct perf *DataUpdate = (struct perf *) malloc(sizeof(struct perf));
  DataUpdate->TTP.Count = 0;
  DataUpdate->TTP.TP = NULL;

  return DataUpdate;
}

//put a packet in the perf structure
void put_packet(struct perf *perf, struct stream *packet, int size)
{
  perf->pack.data = (char *)malloc(size);
  perf->pack.size = size;
  memcpy(perf->pack.data, packet->data, size);
}
```

```
/*create a structure perf, copy the packet
and return the structure created */
struct perf *copypacket(struct stream *packet, int size)
{
  struct perf *DataUpdate = perf_init();
  DataUpdate->pack.data = (char *)malloc(size);
  DataUpdate->pack.size = size;
  memcpy(DataUpdate->pack.data, packet.data, size);
  return DataUpdate;
}

//Sets the time of arrival or departure of the packet
void SetTimePacket(struct perf *DataUpdate, long long int Time)
{
       DataUpdate->time1 = Time;
}


//Sets the type of the packet
void SetTypePacket(struct perf *DataUpdate, unsigned char Type)
{
       DataUpdate->type = Type;
}

//attach the structure perf to the structure listPerf of the peer
void SetDataUpdate(struct peer *peer, struct perf *DataUpdate)
{
       if (peer->UpdateRcvd.Count >= MaxElt)
       {
              AddTopperf(&(peer->UpdateRcvd));
       }
```

```
        peer->UpdateRcvd.HeadPerf->Tab[peer->UpdateRcvd.Count] = DataUpdate;
        peer->UpdateRcvd.Count++;
}

//Add one to Nbr
void AddOneCountPrfTop(struct peer *peer)
{
  peer->UpdateRcvd.HeadPerf->Tab[peer->UpdateRcvd.Count-1]->PrfTop.nbr++;
}

//set the counter of the prefixes
void SetCountPrefTop(struct perf *perf, int Cnt)
{
  perf->PrfTop.nbr = Cnt;
}

//obsoleted by SetCountPrefTop
void SetCountPrfTop(struct peer *peer, int Cnt)
{
  peer->UpdateRcvd.HeadPerf->Tab[peer->UpdateRcvd.Count-1]->PrfTop.nbr = Cnt;
}

//return the first element of the array that is not yet recorded (time1 == -1)
struct Topperf *SeekFirstTimeNonNull(struct listPerf UpdateRcvd, int *CountTmp)
{

  int CountMax = MaxElt;
  struct Topperf *PtrTopperf = UpdateRcvd.HeadPerf, *PtrTopperf2 = NULL;
  *CountTmp = MaxElt;

  while (PtrTopperf->next != NULL)
```

```
        PtrTopperf = PtrTopperf->next;

  if (PtrTopperf->prev == NULL)
    *CountTmp = CountMax = UpdateRcvd.Count;

  while (PtrTopperf != NULL && *CountTmp == CountMax)
  {
    if (PtrTopperf->prev == NULL)
        *CountTmp = CountMax = UpdateRcvd.Count;
    PtrTopperf2 = PtrTopperf;
    while (*CountTmp > 0)
    {
      if (PtrTopperf->Tab[(*CountTmp)-1]->time1 != -1)
        break;
      (*CountTmp)--;
    }
    PtrTopperf = PtrTopperf->prev;
  }
  return PtrTopperf2;
}

/*Creation of a struct Topperf, initializes it and attach it to a struct listPerf */
void AddTopperf(struct listPerf *UpdateRcvd)
{
        struct Topperf *PtrTopperf = UpdateRcvd->HeadPerf;
        UpdateRcvd->HeadPerf = (struct Topperf *)malloc(sizeof(struct Topperf));
        if (PtrTopperf != NULL)
          PtrTopperf->prev = UpdateRcvd->HeadPerf;
        UpdateRcvd->HeadPerf->next = PtrTopperf;
        UpdateRcvd->HeadPerf->prev = NULL;
        UpdateRcvd->Count = 0;
```

```
}

/*************************************************************************/
/************************ FLUSHING the datas *****************************/
/*************************************************************************/

//return the 'sum' of the time-stamps
unsigned long long flush_time_stamp(struct timerPef *TP, int CountTime, unsigned long long Time)
{
  while (CountTime > 0)
  {
    Time -= TP->Time2;
    Time += TP->Time1;
    TP = TP->next;
    CountTime--;
  }
  return Time;
                              }
}

/* flushes all prefixes
returns the 'sum' of the time-stamps due to code for the prefixes */
unsigned long long flush_prefixes(struct pref_eval *PrfEv, int CountPrf, FILE *PtrFi)
{
  unsigned long long Time = 0;
  struct timePerf *TP;
  int CountTime;

  while (CountPrf > 0)
  {
    if (PrfEv != NULL)
```

```
    {
      fwrite(&(PrfEv->len), sizeof(int), 1, PtrFi);
      fwrite(&(PrfEv->val), sizeof(char)*20, 1, PtrFi);
      TP = PrfEv->TTP.TP;
      CountTime = PrfEv->TTP.Count;
      Time = flush_time_stamp(TP, CountTime, Time);

      PrfEv = PrfEv->next;
      CountPrf--;
    }
  }
  return Time;
}

//flushes the information of a peer to file
void flush_peer(struct peer *Peer)
{
    FILE *PtrFi;
    int Cpt, len, CountPrf, CountTime, TotalCount;
    long int lens;
    struct prefix_eval *PrfEv;
    unsigned long long int Time1, Time2;
    struct timePerf *TP;
    struct Topperf *PtrTopperf;

    if ((PtrFi = Peer->UpdateRcvd.PtrFi) != NULL)
    {
        PtrTopperf = Peer->UpdateRcvd.HeadPerf;
        while (PtrTopperf->next != NULL)
           PtrTopperf = PtrTopperf->next;
```

```
while (PtrTopperf != NULL)
{
  if (PtrTopperf->prev == NULL)
    TotalCount = Peer->UpdateRcvd.Count;
  else
    TotalCount = MaxElt;
  for (Cpt = 0; Cpt < TotalCount; Cpt++)
  {
    PrfEv = PtrTopperf->Tab[Cpt]->PrfTop.Prf;
    if (PrfEv != NULL)
    {
      len = strlen(Peer->host);
      fwrite(&len, sizeof(int), 1, PtrFi);
      fwrite(Peer->host, sizeof(char)*len, 1, PtrFi);
      Time1 = PtrTopperf->Tab[Cpt]->time1;

      TP = PtrTopperf->Tab[Cpt]->TTP.TP;
      CountTime = PtrTopperf->Tab[Cpt]->TTP.Count;
      Time1 = flush_time_stamp(TP, CountTime, Time1);

      CountPrf = PtrTopperf->Tab[Cpt]->PrfTop.nbr;
      fwrite(&CountPrf, sizeof(int), 1, PtrFi);
      Time2 = flush_prefixes(PrfEv, CountPrf, PtrFi);

      fwrite(&Time1, sizeof(unsigned long long int), 1, PtrFi);
      fwrite(&Time2, sizeof(unsigned long long int), 1, PtrFi);
      lens = PtrTopperf->Tab[Cpt]->pack.size;
      fwrite(&(lens),sizeof(long int), 1, PtrFi);
      fwrite(PtrTopperf->Tab[Cpt]->pack.data, sizeof(char)*lens, 1, PtrFi);
      free_struct (PtrTopperf->Tab[Cpt]);
    }
```

```
        }
        PtrTopperf = PtrTopperf->prev;
        fflush(PtrFi);
      }
    }
  }

//free the memory allocated for the perf structure
void free_struct(struct perf *Perf)
{
  int CountPrf = --(Perf->PrfTop.nbr), Count2 = Perf->TTP.Count;
  struct prefix_eval *PrfEval, *PrfEvNext;
  struct timePerf *TP;

  PrfEval = Perf->PrfTop.Prf;

  //free the time-stamp for packet
  while (Count2 > 0)
  {
    TP = Perf->TTP.TP->next;
    free(Perf->TTP.TP);
    Perf->TTP.TP = TP;
    Count2--;
  }
  //free structure perf_eval and TP associated with
  while (CountPrf > 0)
    {
        PrfEvNext = PrfEval->next;
        Count2 = PrfEval->TTP.Count;
        while (Count2 > 0)
```

```
                {
                   TP = PrfEval->TTP.TP->next;
                   free(PrfEval->TTP.TP);
                   PrfEval->TTP.TP = TP;
                   Count2--;
                }
                free(PrfEval);
                PrfEval = PrfEvNext;
                CountPrf--;
        }
    free(Perf->pack.data);
    free(Perf);
}



//decrements the semaphore, if equal to 0 then flush the structure
//containing the test information of all peers to file
void DecrementeSema_and_FlushPeer(struct peer *peer)
{
        struct bgp *bgp;
        struct peer *peerTmp;
        struct peer_conf *conf;
        struct listnode *nn;

        OpSema(peer->UpdateRcvd.idsem, 0, -1, 0, 1);
        if (OpSema(peer->UpdateRcvd.idsem, 0, 0, IPC_NOWAIT, 1) != -1)
        {
           nn = peer->conf->head;
           conf = nn->data;
           bgp = conf->bgp;
```

```
          LIST_LOOP(bgp->peer_conf, conf, nn)
          {
            peerTmp = conf->peer;
            flush_peer(peerTmp);
          }
          DelSema(peer->UpdateRcvd.idsem);
        }
}
//return 1 if the packet 's' contains a predefined IP prefix
int writeok(struct stream *s, char *Pref, int Len, int MaskP)
{
  struct bgp_nlri update;
  bgp_size_t withdraw_len, update_len, attribute_len, size;
  u_char *pnt;
  u_char *lim;
  u_char *end;
  struct prefix p;
  int psize;

  memset(&update, 0, sizeof(struct bgp_nlri));

  //jump over the fields in s that are not interesting here
  stream_forward(s, BGP_MARKER_SIZE);
  size = stream_getw(s);
  stream_forward(s, 1);
  end = stream_pnt(s) + size - BGP_HEADER_SIZE;

  withdraw_len = stream_getw(s);
  stream_forward(s, withdraw_len);

  attribute_len = stream_getw(s);
```

```
  stream_forward(s, attribute_len);

  //let the length of the nlri
  update_len = end - stream_pnt(s);

  //if this length != 0 then init the update structure with the nlri of the stream s
  if (update_len)
  {
    update.nlri = stream_pnt(s);
    update.length = update_len;
  }

  pnt = update.nlri;
  lim = pnt+update.length;

  //make the comparison for each prefix in the nlri
  for(; pnt < lim; pnt += psize)
  {
    p.prefixlen = *pnt++;
    psize = PSIZE(p.prefixlen);
    memcpy(&p.u.prefix, pnt, psize);

    if (memcmp(Pref, inet_ntoa(p.u.prefix4), Len) == 0 && p.prefixlen == MaskP)
      return 1;
  }

return 0;
}

/*************************************************************************/
/************************ CHECK END OF TEST ************************/
```

```
/*****************************************************************/

//entry point to compare whether a specified value of community is in the packet s
//return 0 if the community value is found
int CmpCommunity(struct stream *s, u_int32_t community_val)
{
        bgp_size_t withdraw_len, attribute_len, size;
        u_char *end;
        int cp, Res;

        //new structure community initialized with the value commounity_val
        struct community *comm = community_new();
        community_add_val(comm, community_val);

        //set the getp poitner to the begin of the stream s
        //save it to restore it after the function
        cp = stream_get_getp(s);
        stream_set_getp(s, 0);

        //jump over field in s that are not interesting here
        stream_forward(s, BGP_MARKER_SIZE);
        size = stream_getw(s);
        stream_forward(s, 1);

        withdraw_len = stream_getw(s);
        stream_forward(s, withdraw_len);

        attribute_len = stream_getw(s);

        //compare the value of the communities
        Res = comp_community(s, attribute_len, comm);
```

```
        //free the structure community
        community_free(comm);

        //restore the getp pointer
        stream_set_getp(s, cp);
        return Res;
}


//Compare two communities
int comp_community(struct stream *s, bgp_size_t attr_len, struct community *community)
{
  u_char flag;
  u_char type;
  bgp_size_t length;
  u_char *startp, *endp;

  struct community *comm;

  endp = stream_pnt(s) + attr_len;

  /* Get attributes to the end of attribute length. */
  while (stream_pnt(s) < endp)
    {
      /* Fetch attribute flag and type. */
      startp = stream_pnt(s);
      flag = stream_getc (s);
      type = stream_getc (s);

      /* Check extended attribute length bit. */
```

```
      if (CHECK_FLAG (flag, BGP_ATTR_FLAG_EXTLEN))
        length = stream_getw (s);
      else
        length = stream_getc (s);

      /* OK check attribute and check the values. */
      switch (type)
        {
        case BGP_ATTR_COMMUNITIES:
          comm = community_parse (stream_pnt(s), length);
          return community_compare(comm->val, community->val);
          break;
        default:
          stream_forward(s, length);
          break;
        }
    }
  return 1;
}


/*******************************************************************************/
/************************ TIME spent in the code ************************/
/*******************************************************************************/

/* Function getting the number of CPU tick */
inline unsigned long long int GetTick () {
    unsigned long long int x;
    __asm__ volatile (".byte␣0x0f,␣0x31" : "=A" (x)); /* RDTSC */
    // __asm__ volatile ("RDTSC" : "=A" (x));
    return x;
```

```
}

//Add Time1 to struct TimePerf
void AddTime1(struct TopTimePerf *TPPerf, unsigned long long int Time)
{
  struct timePerf *TP = (struct timePerf *)malloc(sizeof(struct timePerf));

  TPPerf->Count++;
  if (TPPerf->TP != NULL)
     TP->next = TPPerf->TP;

  TPPerf->TP = TP;
  TP->Time1 = Time;
  TP->Time2 = 0;
}

//Add Time2 to struct TimePerf
void AddTime2(struct TopTimePerf *TPPerf, unsigned long long int Time)
{
  TPPerf->TP->Time2 = Time;
}

#endif //MODIF_SEB
```

C.1.3 eval_sema.h

```
#ifdef MODIF_SEB

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "PerfEval.h"

key_t CreationCleSema(char *Fi, char any);
int CreationSema(key_t cle, int Nbr);
int SetValeurSema(int idsem, int NumSema, int Valeur);
int OpSema(int idsem, int NumSema, int Op, int flag, unsigned int NbrOp);
int DelSema(int idsem);

#endif //MODIF_SEB
```

C.1.4 eval_sema.c

```
#include "eval_sema.h"

#ifdef MODIF_SEB

/*******************************************************/
//Lib éSmaphore
/*******************************************************/

//creation of the key to use the sema
//return -1 if error
key_t CreationCleSema(char *Fi, char any)
{
       return ftok(Fi, any);
}


//generic creation of Nbr sema
//test first the existence of sema with IPC_EXCL
//if the sema doesn't␣exist␣then␣create␣it
//return the id of sema if all right else -1
int CreationSema(key_t cle, int Nbr)
{
       int idsem;
       //Creation of the semaphore and return of the identifier of the semaphore
       if ((idsem = semget(cle, Nbr, IPC_EXCL)) == -1)
              idsem = semget(cle, Nbr, IPC_CREAT | 0777);
       return idsem;
}
```

```
//put the value of the sema to Valeur
//return -1 if error
int SetValeurSema(int idsem, int NumSema, int Valeur)
{
     return semctl(idsem, NumSema, SETVAL, Valeur);
}

//operator on sema
int OpSema(int idsem, int NumSema, int Op, int flag, unsigned int NbrOp)
{
     struct sembuf op = {NumSema, Op, IPC_NOWAIT};
     return semop(idsem, &op, NbrOp);
}

//Destruction of sema
int DelSema(int idsem)
{
     return semctl(idsem, 0, IPC_RMID, 0);
}

#endif //MODIF_SEB
```

C.2 Code de l'amélioration

C.2.1 bgp_packet.h

```
/*
 * BGP packet management header.
 * Copyright (C) 1999 Kunihiro Ishiguro
 *
 * This file is part of GNU Zebra.
 *
 * GNU Zebra is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation; either version 2, or (at your option) any
 * later version.
 *
 * GNU Zebra is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Zebra; see the file COPYING. If not, write to the Free
 * Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
 * 02111-1307, USA.
 */

#ifndef _ZEBRA_BGP_PACKET_H
#define _ZEBRA_BGP_PACKET_H

//int writeok(struct stream *, char Pref);
/* Packet send and receive function prototypes. */
```

```
int bgp_read (struct thread *);
int bgp_write (struct thread *);

void bgp_keepalive_send (struct peer *);
void bgp_open_send (struct peer *);
void bgp_notify_send (struct peer *, u_char, u_char);
void bgp_notify_send_with_data (struct peer *, u_char, u_char, u_char *, size_t);
void bgp_update_send (struct peer_conf *, struct peer *, struct prefix *, struct attr *, u_int16_t, u_char,
       struct peer *, struct prefix_rd *, u_char *);
void bgp_withdraw_send (struct peer *, struct prefix *, afi_t, safi_t, struct prefix_rd *, u_char *);
void bgp_route_refresh_send (struct peer *, afi_t, safi_t);

#ifdef MULT_UPDATE
void bgp_packet_withdraw_clear(struct peer *peer);
void bgp_withdraw_build(struct peer *peer, struct prefix *p, afi_t afi, safi_t safi,
          struct prefix_rd *prd, u_char *tag);
void bgp_update_build(struct peer_conf *conf, struct peer *peer,
       struct prefix *p, struct attr *attr, afi_t afi, safi_t safi,
       struct peer *from, struct prefix_rd *prd, u_char *tag);
struct stream *bgp_packet_update_init(struct peer *peer, struct attr *attr);
void bgp_packet_put_update_fifo(struct peer *peer, struct stream *s);
void bgp_packet_update_clear(struct peer *peer);
#ifdef OPT_SIZE_UPDATE
void clear_to_announce_hash(struct peer *peer);
#include "hash.h"
#endif //OPT_SIZE_UPDATE
#endif //MULT_UPDATE
#endif /* _ZEBRA_BGP_PACKET_H */
```

C.2.2 bgp_packet.c

```
#ifdef MULT_UPDATE
#include "PerfEval.h"
#include "flush_peer.h"

//Init an update message
struct stream *bgp_packet_update_init(struct peer *peer, struct attr *attr)
{

 struct stream *s;
 char attrstr[BUFSIZ];

 /* Make attribute dump string. */
 if (attr != NULL)
   bgp_dump_attr (peer, attr, attrstr, BUFSIZ);

 s = stream_new (BGP_MAX_PACKET_SIZE);

 /* Make BGP update packet. */
 bgp_packet_set_marker (s, BGP_MSG_UPDATE);

 /* Unfeasible Routes Length. */
 stream_putw (s, 0);

 return s;
}

void bgp_packet_put_attribute(struct peer_conf *conf, struct peer *peer,
  struct prefix *p, struct attr *attr, afi_t afi, safi_t safi,
```

```
  struct peer *from, struct prefix_rd *prd, u_char *tag, struct stream *s)
{
 unsigned long pos;
 bgp_size_t total_attr_len;

 /* Make place for total attribute length. */
 pos = stream_get_putp (s);
 stream_putw (s, 0);

 //TO DO to support the multiple prefix in the MP_REACH_NLRI, this function must be modified
 //insert attributes in the message
 total_attr_len = bgp_packet_attribute (conf, peer, s, attr, p, afi, safi, from, prd, tag);

 /*Set Total Path Attribute Length. */
 stream_putw_at (s, pos, total_attr_len);

}

void bgp_set_withdraw_length(struct stream *s)
{
  unsigned long pos;
  bgp_size_t unfeasible_len;

  /* save the position in the stream */
  pos = stream_get_putp(s);

  /* place the putp pointer at the withdraw length field */
  stream_set_putp(s, BGP_MARKER_SIZE+3);

  unfeasible_len = pos - stream_get_putp(s) - 2;
```

```
  /* Set unfeasible len. */
  stream_putw (s, unfeasible_len);

  /* restore the position */
  stream_set_putp(s, pos);

  /* Set total path attribute length. */
  stream_putw (s, 0);
}

void bgp_withdraw_build(struct peer *peer, struct prefix *p, afi_t afi, safi_t safi,
          struct prefix_rd *prd, u_char *tag)
{
  struct stream *s;
  unsigned long pos;
  bgp_size_t total_attr_len;

  /* no msg in construction so init one */
  if (peer->withdraw_msg == NULL)
       peer->withdraw_msg = bgp_packet_update_init(peer, NULL);

  /* IPv4 */
  if (p->family == AF_INET && safi == SAFI_UNICAST)
  {
    /* if enough places insert prefix else send it and re-call it */
    if (stream_get_putp(peer->withdraw_msg) + PSIZE(p->prefixlen) + 1 < BGP_MAX_PACKET_SIZE - 2)
        stream_put_prefix(peer->withdraw_msg, p);
    else
    {
        /* set the length of the withdraw NLRI field */
        bgp_set_withdraw_length(peer->withdraw_msg);
```

```
        bgp_packet_put_update_fifo(peer, peer->withdraw_msg);
        stream_free(peer->withdraw_msg);
        peer->withdraw_msg = NULL;
        bgp_withdraw_build(peer, p, afi, safi, prd, tag);
    }
 }
 /* MP_UNREACH_NLRI */
 else
 {
        s = bgp_packet_update_init(peer, NULL);

        pos = stream_get_putp (s);

        stream_putw (s, 0);
        total_attr_len = bgp_packet_withdraw (peer, s, p, afi, safi, prd, tag);

        /* Set total path attribute length. */
        stream_putw_at (s, pos, total_attr_len);

        bgp_packet_put_update_fifo(peer, peer->withdraw_msg);

        stream_free(s);
 }
}

//Add a prefix to update message
//if it's␣a␣prefix␣that␣is␣inserted␣in␣the␣MP_REACH_NLRI,␣it␣sends␣it␣immediately␣after
//the construction
void bgp_update_build(struct peer_conf *conf, struct peer *peer,
        struct prefix *p, struct attr *attr, afi_t afi, safi_t safi,
        struct peer *from, struct prefix_rd *prd, u_char *tag)
```

```
{
 struct stream *s;
 struct to_announce *to_announce;
 unsigned long putp;

#ifdef DISABLE_BGP_ANNOUNCE
 return;
#endif /* DISABLE_BGP_ANNOUNCE */

#ifdef OPT_SIZE_UPDATE
  to_announce = to_announce_intern(peer->to_announce_hash, peer->list_ta, attr);
  s = to_announce->stream;
#else
  to_announce = to_announce_init(bgp_attr_intern(attr));
  if (peer->to_announce != NULL)
  {
    if (to_announce_cmp(peer->to_announce, to_announce) == 0)
    {
      bgp_packet_put_update_fifo(peer, peer->to_announce->stream);
      stream_free(peer->to_announce->stream);
      peer->to_announce = to_announce;
      s = NULL;
    }
    else
    {
      bgp_attr_unintern(attr);
      free(to_announce);
      to_announce = peer->to_announce;
      s = to_announce->stream;
    }
  }
```

```
  else
  {
    s = to_announce->stream;
    peer->to_announce = to_announce;
  }
#endif //OPT_SIZE_UPDATE
  if (!s)
  {
    s = bgp_packet_update_init(peer, attr);
    bgp_packet_put_attribute(conf, peer, p, attr, afi, safi, from, prd, tag, s);
    to_announce->stream = s;
  }

    //if the size of the update + the size of the new prefix is less than the MAX SIZE of a packet
    //then put this prefix if it's␣a␣IPv4␣address.␣else␣send␣it␣!
    //else send the packet and re-call this function
    putp = s->putp;
      /* NLRI set. */
    if (p->family == AF_INET && safi == SAFI_UNICAST)
    {
      if (putp + PSIZE(p->prefixlen) + 1 <= BGP_MAX_PACKET_SIZE)
        stream_put_prefix (s, p);
      else
      {
        bgp_packet_put_update_fifo(peer, s);
        stream_free(s);
        to_announce->stream = NULL;
        bgp_update_build(conf, peer, p, attr, afi, safi, from, prd, tag);
      }
    }
    else //it must be there until the MultiProtocol with multiple routes in the MP_REACH_NLRI is supported
```

```
    {
      bgp_packet_put_update_fifo(peer, s);
      stream_free(s);
      to_announce->stream = NULL;
    }
}

//put the packet in the FIFO file to send it
//sets the size of this packet too
void bgp_packet_put_update_fifo(struct peer *peer, struct stream *s)
{
  struct stream *packet;

 /* Set size. */
 bgp_packet_set_size (s, 0);

 packet = bgp_packet_dup (s);

 /* Dump packet if debug option is set. */
#ifdef DEBUG
 bgp_packet_dump (packet);
#endif /* DEBUG */

 /* Add packet to the peer. */
 bgp_packet_add (peer, packet);

 BGP_WRITE_ON (peer->t_write, bgp_write, peer->fd);
}

//sends the last packet(s) of a peer
void bgp_packet_update_clear(struct peer *peer)
```

```
{
  struct to_announce *to_announce, *to_announce_prev;

#ifdef OPT_SIZE_UPDATE
  to_announce = peer->list_ta->first;
  peer->list_ta->first = NULL;
  peer->list_ta->last = NULL;
  peer->list_ta->count = 0;
#else
  to_announce = peer->to_announce;
  peer->to_announce = NULL;
#endif //OPT_SIZE_UPDATE

  while (to_announce != NULL)
  {
    if (to_announce->stream != NULL)
    {
      bgp_packet_put_update_fifo(peer, to_announce->stream);
      stream_free(to_announce->stream);
      to_announce->stream = NULL;
    }
    to_announce_prev = to_announce;
    bgp_attr_unintern(to_announce->attr);
    to_announce = to_announce->next;
    free(to_announce_prev);
  }

#ifdef OPT_SIZE_UPDATE
  clear_to_announce_hash(peer);
#endif //OPT_SIZE_UPDATE
}
```

```
#ifdef OPT_SIZE_UPDATE

/* sets to null each element of hash->index */
void clear_to_announce_hash(struct peer *peer)
{
  struct Hash *ta_hash = peer->to_announce_hash;
  int i = 0;
  while (i < HASHTABSIZE)
  {
    ta_hash->index[i] = NULL;
    i++;
  }
}
#endif //OPT_SIZE_UPDATE

/* sends the last message with withdraw NLRI */
void bgp_packet_withdraw_clear(struct peer *peer)
{
  if (peer->withdraw_msg != NULL)
  {
    bgp_packet_put_update_fifo(peer, peer->withdraw_msg);
    stream_free(peer->withdraw_msg);
    peer->withdraw_msg = NULL;
  }
}
#endif //MULT_UPDATE
```

C.2.3 bgp_to_announce.h

```
#ifndef TO_ANNOUNCE_H
#define TO_ANNOUNCE_H
#ifdef MULT_UPDATE

#ifdef MUP_SEND
#include "flush_peer.h"
#endif //MUP_SEND
struct to_announce
{
  struct attr *attr;
  struct stream *stream;
  struct to_announce *next;
#ifdef MUP_SEND
  struct perf *perf;
#endif //MUP_SEND
};

#ifdef OPT_SIZE_UPDATE
struct list_to_announce
{
  int count;
  struct to_announce *first;
  struct to_announce *last;
};
struct to_announce *to_announce_intern(struct Hash *ta_hash, struct list_to_announce *list_ta, struct attr *attr);
struct Hash *to_announce_hash_init ();
void delete_to_announce_hash(struct peer *peer);
#endif //OPT_SIZE_UPDATE
```

```
unsigned int to_announce_key_make(struct to_announce *to_announce);
int to_announce_cmp(struct to_announce *ta1, struct to_announce *ta2);
struct to_announce *to_announce_init(struct attr *attr);
void bgp_update_delete(struct peer *peer);
void bgp_withdraw_delete(struct peer *peer);
#endif //MULT_UPDATE
#endif //TO_ANNOUNCE_H
```

C.2.4 bgp_to_announce.c

```
#include <zebra.h>

#ifdef MULT_UPDATE

#include "hash.h"
#include "vty.h"
#include "prefix.h"

#include "bgpd/bgpd.h"
#include "bgpd/bgp_attr.h"
#include "bgpd/bgp_community.h"
#include "bgpd/bgp_ecommunity.h"
#include "bgpd/bgp_aspath.h"

#include "bgpd/bgp_to_announce.h"
#define MAX_HASHTABSIZE 2048

extern unsigned int cluster_hash_key_make (struct cluster_list *cluster);
extern unsigned int transit_hash_key_make (struct transit *transit);
extern int cluster_hash_cmp (struct cluster_list *cluster1, struct cluster_list *cluster2);
extern int community_cmp (struct community *com1, struct community *com2);
extern int ecommunity_cmp (struct ecommunity *ecom1, struct ecommunity *ecom2);

//creation of a hash key
unsigned int to_announce_key_make(struct to_announce *to_announce)
{
  unsigned long int key = 0;
#ifdef HAVE_IPV6
```

```
  int i;
#endif //HAVE_IPV6
  struct attr *attr = to_announce->attr;

  key += attr->origin;
  key += attr->nexthop.s_addr;
  key += attr->med;
  key += attr->local_pref;
  key += attr->aggregator_as;
  key += attr->aggregator_addr.s_addr;

#ifdef HAVE_IPV6
  key += attr->mp_nexthop_len;
  for (i = 0; i < 16; i++)
    key += attr->mp_nexthop_global.s6_addr[i];
  for (i = 0; i < 16; i++)
    key += attr->mp_nexthop_local.s6_addr[i];
#endif /* HAVE_IPV6 */

  key += attr->mp_nexthop_global_in.s_addr;
  if (attr->aspath)
    key += aspath_key_make (attr->aspath);

  if (attr->community)
    key += community_hash_make (attr->community);

  if (attr->ecommunity)
    key += ecommunity_hash_make (attr->ecommunity);

   if (attr->cluster)
    key += cluster_hash_key_make (attr->cluster);
```

```
  if (attr->transit)
    key += transit_hash_key_make (attr->transit);

 return (unsigned int)key %= MAX_HASHTABSIZE;
}

//compares two to_announce structures (used for the hash table)
//return 0 if different, 1 if equal
int to_announce_cmp(struct to_announce *ta1, struct to_announce *ta2)
{
  struct attr *attr1, *attr2;
  attr1 = ta1->attr;
  attr2 = ta2->attr;

  //we may check the pointer of attr1 and attr2 to check if they are the same
  //But if the address aren't␣the␣same␣it's not a reason to think they don't␣share␣the
  //same attributes 'cause␣there␣is␣the␣WEIGHT␣attribute␣in␣this␣structure␣and␣there␣is␣no
  //need to have the both equals !!!!
  if (attr1 == attr2)
    return 1;
  else
  {
    if (attr1->flag == attr2->flag)
    {

        //check for the origin attribute
        if (attr1->origin != attr2->origin)
          return 0;

        //check for the local_pref attribute
```

```
  if (attr1->local_pref != attr2->local_pref)
    return 0;

  //check for the aspath attribute
  if (attr1->aspath != attr2->aspath)
    return 0;

  //check for the equality of the nexthop
  if (!IPV4_ADDR_SAME(&(attr1->nexthop), &(attr2->nexthop)))
    return 0;

//check if there is a MED attribute
if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_MULTI_EXIT_DISC))
  if (attr1->med != attr2->med)
    return 0;

/*check if there is an ATOMIC-AGGREGATE attribute -> we don't care of this,
  it's already done in the check of the equality of the attributes */
//if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_ATOMIC_AGGREGATE))

//check if there is an AGGREGATOR attribute
if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_AGGREGATOR))
  if (attr1->aggregator_as != attr2->aggregator_as
      || !IPV4_ADDR_SAME(&(attr1->aggregator_addr), &(attr2->aggregator_addr)))
    return 0;

//check if there is a COMMUNITY attribute
if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_COMMUNITIES))
  if (community_cmp(attr1->community, attr2->community) == 0)
    return 0;
```

```
      //check if there is an ORIGINATOR-ID attribute
      if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_ORIGINATOR_ID))
        if (!IPV4_ADDR_SAME(&(attr1->originator_id), &(attr2->originator_id)))
          return 0;

      //check if there is a CLUSTER_LIST attribute
      if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_CLUSTER_LIST))
        if (cluster_hash_cmp(attr1->cluster, attr2->cluster))
          return 0;

      //checks for the MultiProtocol attributes
#ifdef HAVE_IPV6
       if (attr1->mp_nexthop_len != attr2->mp_nexthop_len)
        return 0;

       if (!IPV6_ADDR_SAME (&attr1->mp_nexthop_global, &attr2->mp_nexthop_global))
        return 0;

       if (!IPV6_ADDR_SAME (&attr1->mp_nexthop_local, &attr2->mp_nexthop_local))
        return 0;

#endif //HAVE_IPV6

      //check for the equality of the mp_* attributes
      if (!IPV4_ADDR_SAME (&attr1->mp_nexthop_global_in, &attr2->mp_nexthop_global_in))
          return 0;

      //check if there is a EXTENDED-COMMUNITY attribute
      if (attr1->flag & ATTR_FLAG_BIT (BGP_ATTR_EXT_COMMUNITIES))
        if (ecommunity_cmp(attr1->ecommunity, attr2->ecommunity) == 0)
          return 0;
```

```
      //before checking the equality of the transit values we have to
      //check if there are values in both
      if (attr1->transit && attr2->transit
      && (attr1->transit->length == attr2->transit->length))
      {
         if (memcmp(attr1->transit->val, attr2->transit->val, attr1->transit->length) == 1)
           return 0;
      }
      return 1;
    }
  }
  return 0;
}

//initialization of a structure to_announce
struct to_announce *to_announce_init(struct attr *attr)
{
  struct to_announce *to_announce = (struct to_announce *)malloc(sizeof(struct to_announce));

  to_announce->attr = attr;
  to_announce->stream = NULL;
  to_announce->next = NULL;

  return to_announce;
}

#ifdef OPT_SIZE_UPDATE

//Put the attr in a to_announce structure and then put it in the hash table if it is not already in
struct to_announce *to_announce_intern(struct Hash *ta_hash, struct list_to_announce *list_ta, struct attr *attr)
```

```
{
    struct to_announce *to_announce, *ta_to_search;
    int found = 1;

    ta_to_search = to_announce_init(bgp_attr_intern(attr));

   //if the structure attr is not interned in the hash table
   //push it in the hashtable
   //but also put it in the link list for list_to_announce
   if ((to_announce = (struct to_announce *)hash_search(ta_hash, ta_to_search)) == NULL)
   {
        found = 0;
        to_announce = (struct to_announce *)(hash_push (ta_hash, ta_to_search))->data;

        //ta_to_search is now equal to the last element of the link list of to_announce
        ta_to_search = list_ta->last;

        /* if there is an element in the link list
             the new last element is the new announce so put it right after the old last element
           else
             put the new to_announce it's the first element */
        if (ta_to_search != NULL)
          ta_to_search->next = to_announce;
        else
          list_ta->first = to_announce;

          //in all the cases it's␣the␣new␣last␣element
          list_ta->last = to_announce;
          list_ta->count++;

   }
```

```
   /*we have to free the struct to_announce created for the search
     if we have already a struct to_announce in our hash table the flag found will be
     to 1 we can also unintern the attr */
   if (found)
   {
     bgp_attr_unintern(attr);
     free(ta_to_search);
   }
    return to_announce;
}

/*init of the hashtable with the functions
the first is the function of comparison
the second is the function calculating the key based on the attributes */
struct Hash *to_announce_hash_init ()
{
 struct Hash *ta_hash;
 ta_hash = hash_new (MAX_HASHTABSIZE);
 ta_hash->hash_key = to_announce_key_make;
 ta_hash->hash_cmp = to_announce_cmp;

 return ta_hash;
}
#endif //OPT_SIZE_UPDATE

/* deletes the packet(s) of a peer */
void bgp_update_delete(struct peer *peer)
{
  struct to_announce *to_announce, *to_announce_prev;

#ifdef OPT_SIZE_UPDATE
```

```
  if (peer->list_ta != NULL)
  {
    to_announce = peer->list_ta->first;
    peer->list_ta->first = NULL;
    peer->list_ta->last = NULL;
    peer->list_ta->count = 0;
  }
  else
    to_announce = NULL;
#else
  to_announce = peer->to_announce;
  peer->to_announce = NULL;
#endif //OPT_SIZE_UPDATE

  while (to_announce != NULL)
  {
    if (to_announce->stream != NULL)
    {
      stream_free(to_announce->stream);
      to_announce->stream = NULL;
    }
    to_announce_prev = to_announce;
    bgp_attr_unintern(to_announce->attr);
    to_announce = to_announce->next;
    free(to_announce_prev);
  }

#ifdef OPT_SIZE_UPDATE
  delete_to_announce_hash(peer);
#endif //OPT_SIZE_UPDATE
}
```

```
#ifdef OPT_SIZE_UPDATE

/* deletes the hash table of a peer */
void delete_to_announce_hash(struct peer *peer)
{
  if (peer->to_announce_hash != NULL)
  {
    XFREE(MTYPE_HASH, peer->to_announce_hash->index);
    XFREE(MTYPE_HASH, peer->to_announce_hash);
  }
}
#endif //OPT_SIZE_UPDATE

/* deletes the withdraw msg of a peer */
void bgp_withdraw_delete(struct peer *peer)
{
  if (peer->withdraw_msg != NULL)
  {
    stream_free(peer->withdraw_msg);
    peer->withdraw_msg = NULL;
  }
}
#endif //MULT_UPDATE
```